# DataMing HW3

姓名 : 吳建澄
學號 : NM6111035

、HITS

1. 概述：

HITS 是可以用於幫網頁評分的　種 link analysis algorithm, 它的算法主要是計算兩個數值－Hub, Authority，這 兩種數值會互相影響，可以看到 Hub, Authority 的算法如下圖, Hub 是所有 v node's children 之 Authority 總和, Authority 是所有 v node's parents 之 Hub 總和。

- Recursive dependency:

$$a(v) \leftarrow \sum_{w \in pa[v]} h(w)$$

$$h(v) \leftarrow \sum_{w \in ch[v]} a(w)$$

透過上式, 不斷重複計算每個 node 的 Hub, Authority, 最終會 Converge 在個數值。

2. 程式碼簡述：

i. Node class:

這個class主要是用來建立graph的一個基本單位, 總共有name、children、parents、auth、preAuth、hub、preHub attribute。

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.parents = []
        self.auth = 1.0
        self.preAuth = 1.0
        self.hub = 1.0
        self.preHub = 1.0
```

ii. Graph class:

這個class主要包含建立graph一些function, 包含增加邊, 以及尋找Node的function。

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

    def search(self, name):

        exist = False

        for node in self.Nodes:
            if(node.name == name):
                exist = True
                break

        if exist:
            return next(node for node in self.Nodes if node.name == name)

        else:
            new_node = Node(name)
            self.Nodes.append(new_node)
            return new_node

    def addEdge(self, parent, child):
        parent_node = self.search(parent)
        child_node = self.search(child)

        if(child_node.name not in parent_node.children):
            parent_node.children.append(child_node)

        if(parent_node.name not in child_node.parents):
            child_node.parents.append(parent_node)
```

1. addEdge function

和 function 名稱意思相同, 將 parent 和 child 兩個 node連結起來。

2. search function

和 function 名稱意思相同, 用來找尋 graph 中想找到node, 若無該 node, 則建立　個新的 node, 以便後續建立連結。

iii. HIT:

經由HIT algorithm, 下圖為我的程式碼對應片段

```
def HITS(g, num):
    for i in range(num):
        nodeList = g.Nodes
        for node in nodeList:
            node.auth = sum(parent_node.preHub for parent_node in node.parents)

        for node in nodeList:
            node.hub = sum(child_node.preAuth for child_node in node.children)

        auth_sum = sum(node.auth for node in nodeList)
        hub_sum = sum(node.hub for node in nodeList)

        for node in nodeList:
            node.auth /= auth_sum
            node.hub /= hub_sum
            node.f = node.auth
            node.preHub = node.hub
```

1. 將每個 node 的 parent(parent_node)的 Hub 加總, 成為該節點的新 Authority 。
2. 計算目前每個節點的 Authority 總和, 用於 normalize 歸一化。
3. 將每個節點的 Authority 進行 normalize。

以上是Authority的部分, Hub部分與其類似。

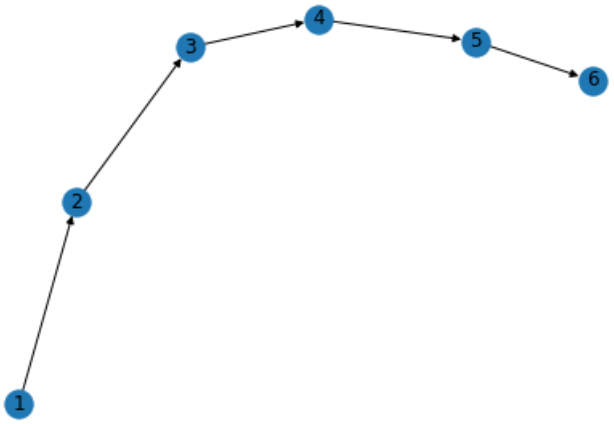
iv. Main:

主程式如下, file_path 決定要分析的檔案, it決定iteration的次數, 其他就是建立Graph以及使用HIT function來完成此次任務。

```
if __name__ == '__main__':
    authority_fname = '_HITS_authority.txt'
    hub_fname = '_HITS_hub.txt'
    result_dir = 'result'
    data_path = './hw3dataset/'
    for file_name in os.listdir(data_path):

        file_path = data_path + file_name
        fname = file_path.split('/')[-1].split('.')[0]
        it = 30
        graph = init_graph(file_path)
        start = time.time()
#         graph.display()
        HITS(graph, it)
        end = time.time()
        print(end-start)

        auth_list, hub_list = get_auth_hub_list(graph)
        print()
        print(fname)
        print('Authority:')
        print(auth_list)
        path = os.path.join(result_dir, fname)
        os.makedirs(path, exist_ok=True)
        np.savetxt(os.path.join(path, fname + authority_fname), auth_list, fmt='%.3f', newline=" ")
        print('Hub:')
        print(hub_list)
        print()
        np.savetxt(os.path.join(path, fname + hub_fname), hub_list, fmt='%.3f', newline=" ")
```

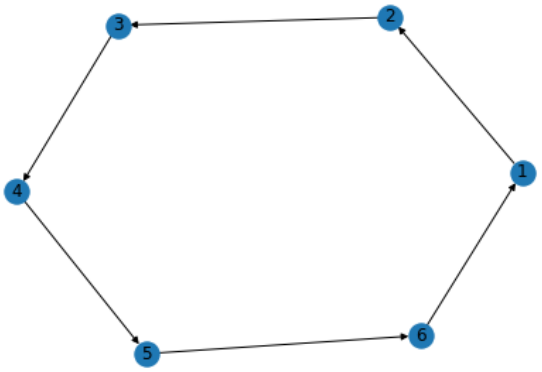
3. 結果

i. graph_1

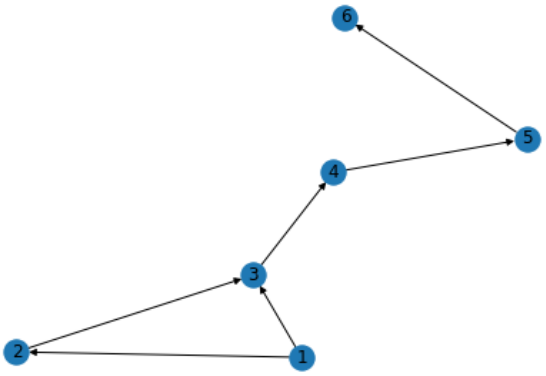



```
graph_1
Authority:
[0.  0.2 0.2 0.2 0.2 0.2]
Hub:
[0.2 0.2 0.2 0.2 0.2 0. ]
```

上圖，可以看到 graph_1 節點間的關係，我們可以探討 Node 1 的 Authority, Hub 值。由 HITS algorithm 可以得知 Authority 是其父節點的 Hub 加總，但因為 Node 1 並沒有 parent nodes，因此其 Authority為 0，若想增加 Node 1 的 Authority，可以透過將其餘 4 個node 中其中　個 node 指向 Node 1，但相對 Hub 卻下降(如下圖，node 6 指向 node 1)。另外是 Node 1 的 Hub 值，從公式中可以看出 Hub 是由其子節點的 Authority 加總，因此若想增加 Node 1 的 Hub，可以增加 Node 1 的 childnode(如下圖二，新增 node 1 指向 node 3)，亦或是使其child node 的 Authority 增加。

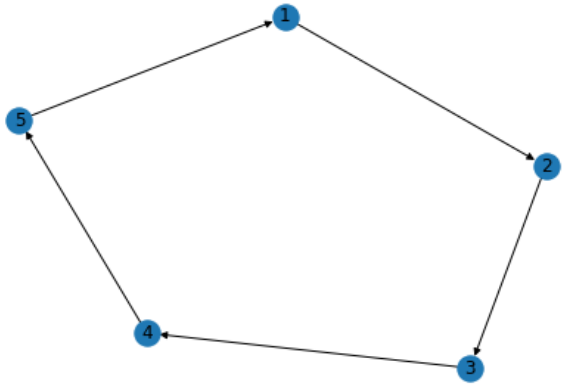



```
graph_1
Authority:
[0.167 0.167 0.167 0.167 0.167 0.167]
Hub:
[0.167 0.167 0.167 0.167 0.167 0.167]
```



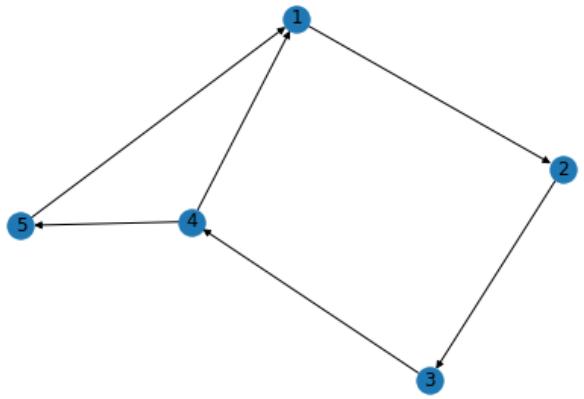

```
graph_1
Authority:
[0.    0.25  0.375 0.125 0.125 0.125]
Hub:
[0.333 0.167 0.167 0.167 0.167 0.   ]
```

ii. graph_2

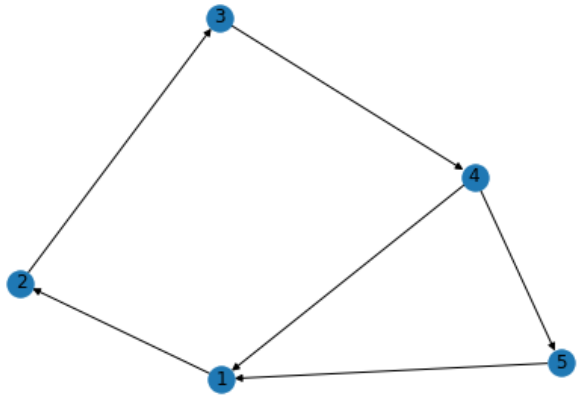



```
graph_2
Authority:
[0.2 0.2 0.2 0.2 0.2]
Hub:
[0.2 0.2 0.2 0.2 0.2]
```

上圖，可以看到 graph_2 節點間的關係，我們可以探討Node 1 的 Authority, Hub 值。由 HITS algorithm 可以得知 Authority 是其父節點的 Hub 加總，若想增加 Node 1 的 Authority，可以透過增加 node 1 的父節點(如下圖，node 4 指向 node 1)。另外是 Node 1 的Hub 值，從公式中可以看出 Hub 是由其子節點的 Authority加總，因此若想增加 Node 1 的 Hub，可以增加 Node 1 的child node(如下圖，新增 node 1 指向 node 4)，亦或是使其 child node 的 Authority 增加。

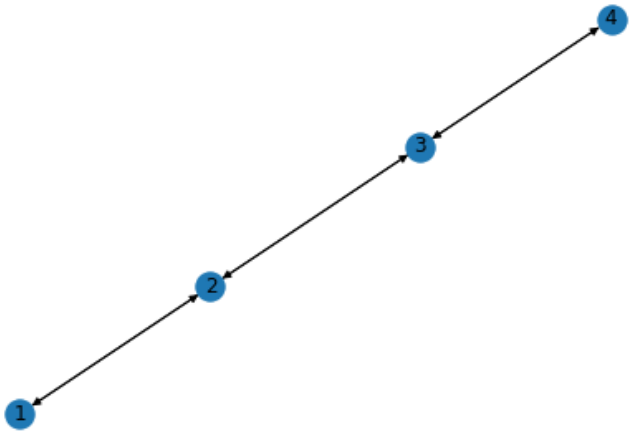



```
graph_2
Authority:
[0.375 0.125 0.125 0.125 0.25 ]
Hub:
[0.167 0.167 0.167 0.333 0.167]
```



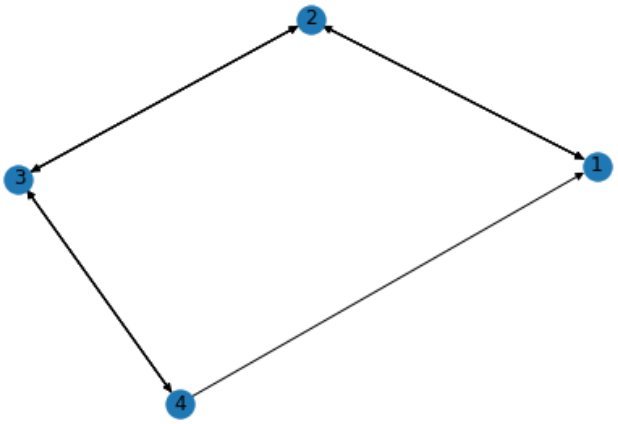

```
graph_2
Authority:
[0.125 0.25  0.125 0.375 0.125]
Hub:
[0.333 0.167 0.167 0.167 0.167]
```

iii. graph_3

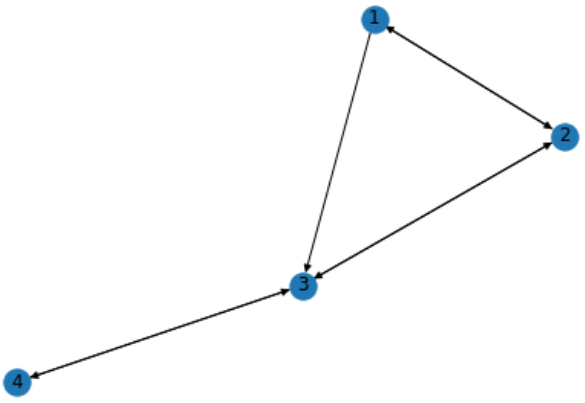



```
graph_3
Authority:
[0.2 0.3 0.3 0.2]
Hub:
[0.167 0.333 0.333 0.167]
```

上圖，可以看到 graph_3 節點間的關係，我們可以探討Node 1 的 Authority, Hub 值。由 HITS algorithm 可以得知 Authority 是其父節點的 Hub 加總，若想增加 Node 1 的 Authority，可以透過增加 node 1 的父節點(如下圖，node 4 指向 node 1)。另外是 Node 1 的Hub 值，從公式中可以看出 Hub 是由其子節點的 Authority加總，因此若想增加 Node 1 的 Hub，可以增加 Node 1 的child node(如下圖，新增 node 1 指向 node 3)，亦或是使其 child node 的 Authority 增加。




```
graph_3
Authority:
[0.308 0.231 0.308 0.154]
Hub:
[0.143 0.286 0.286 0.286]
```




```
graph_3
Authority:
[0.154 0.308 0.385 0.154]
Hub:
[0.286 0.286 0.286 0.143]
```

# 二、PageRank

## 1. 概述:

PageRank 是能夠被應用在網頁排名的　種 link analysisalgorithm，它的核心概念是評分越高的 node 通常被許多其他node 所連結(或 reference), 評分的算法為下圖。

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

D(damping factor)=0.1~0.15
n=|page set|

它的算法是計算評分數值，是所有 v node's parents 之評分/其children 數後的總和。透過上式, 不斷重複計算每個 node 的評分, 最終會Converge 在　個數值。**阻尼係數(damping factor)所代表的意義為, 用戶訪問到某頁面後繼續訪問下一個頁面的概率, 相對應的 (1-d) 則是用戶停止點擊, 隨機瀏覽新網頁的概率。的大小由一般上網者使用瀏覽器書籤功能的頻率的平均值估算得到。**

## 2. 程式碼簡述:

### i. Node class:

這個class主要是用來建立graph的一個基本單位, 總共有name、children、parents、rank attribute。

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.parents = []
        self.rank = 1.0
```

### ii. Graph class:

這個class主要包含建立graph一些function, 包含增加邊, 以及尋找Node的function。(這部份的程式碼和HIT演算法所使用的方式一模一樣)

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

    def search(self, name):

        exist = False

        for node in self.Nodes:
            if(node.name == name):
                exist = True
                break

        if exist:
            return next(node for node in self.Nodes if node.name == name)

        else:
            new_node = Node(name)
            self.Nodes.append(new_node)
            return new_node

    def addEdge(self, parent, child):
        parent_node = self.search(parent)
        child_node = self.search(child)

        if(child_node.name not in parent_node.children):
            parent_node.children.append(child_node)

        if(parent_node.name not in child_node.parents):
            child_node.parents.append(parent_node)
```

1. addEdge function

和 function 名稱意思相同, 將 parent 和 child 兩個 node連結起來。

2. search function

和 function 名稱意思相同, 用來找尋 graph 中想找到node, 若無該 node, 則建立　個新的 node, 以便後續建立連結。

iii. PageRank function:

經由 PageRank algorithm, 下圖為我的程式碼對應片段

```
def PageRank(g, dampingFactor, num):
    for i in range(num):
        nodeList = g.Nodes
        for node in nodeList:
            pNodes = node.parents
            pageRankSum = sum((pNode.rank / len(pNode.children)) for pNode in pNodes)
            node.rank = (dampingFactor / len(g.Nodes)) + (1-dampingFactor) * pageRankSum

        pageRankSum = sum(node.rank for node in g.Nodes)
        for node in g.Nodes:
            node.rank /= pageRankSum
```

由上圖可以將我的程式碼分成 2 個部分

1. 計算 PageRank 的 sum

2. 計算當前 node 的 pageRank
3.將 PageRank 進行 normalize 處理

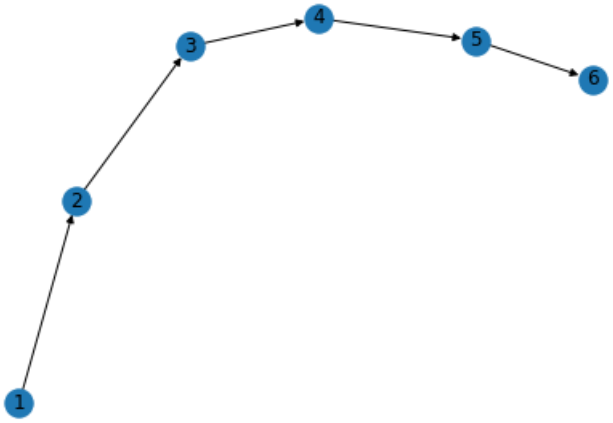
### iv. Main:

主程式如下，file_path 決定要分析的檔案，it決定iteration的次數，Damping Factor 設定為0.1。其他就是建立Graph以及使用PageRank function來完成此次任務。

```
import os
import time
if __name__ == '__main__':

    iteration = 30
    dampingFactor = 0.1
    data_path = './hw3dataset/'
    for file_name in os.listdir(data_path):

        file_path = data_path + file_name
        result_dir = 'result'
        fname = file_path.split('/')[-1].split('.')[0]
        pagerank_fname = '_PageRank.txt'

        graph = init_graph(file_path)

        start = time.time()

        PageRank(graph, dampingFactor, iteration)
        end = time.time()
        print(end-start)
        pagerank_list = get_pagerank_list(graph)
        print('PageRank:')
        print(pagerank_list)
        print()
        path = os.path.join(result_dir, fname)
        os.makedirs(path, exist_ok=True)
        np.savetxt(os.path.join(path, fname + pagerank_fname), pagerank_list, fmt='%.3f', newline=" ")
```

## 3.結果

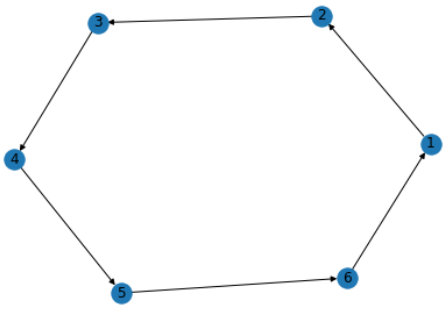
### i. graph_1(Damping Factor = 0.1)




```
graph_1.txt
PageRank:
[0.056 0.107 0.152 0.193 0.23  0.263]
```

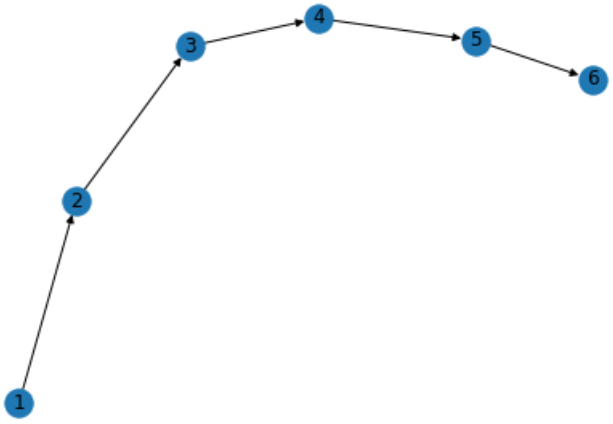
上圖，可以看到 graph_1 節點間的關係，我們可以探討Node 1 的 PageRank 值。由 PageRank algorithm 可以得知 PageRank 它的核心概念是評分越高的

node 通常被許多其他 node 所連結(或reference), 若想增加 Node 1 的 PageRank, 可以透過將其餘4 個 node 中其中　個 node 指向 Node 1 (如下圖, node 6指向 node 1)。




```
graph_1.txt
PageRank:
[0.167 0.167 0.167 0.167 0.167 0.167]
```

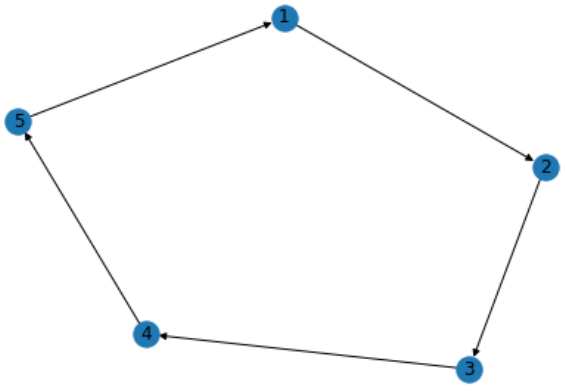
ii. graph_1(Damping Factor = 0.15)




```
graph_1.txt
PageRank:
[0.061 0.112 0.156 0.193 0.225 0.252]
```

由上圖可以發現增加了damping factor , PageRank 原本較低的值都變高了; 相反的, 原本較高的值都變低了。

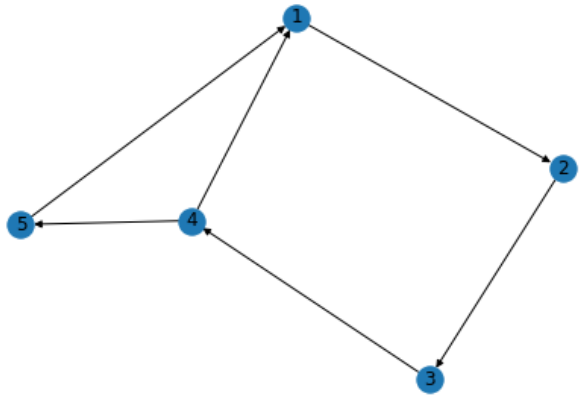
iii. graph_2(Damping Factor = 0.1)




```
graph_2.txt
PageRank:
[0.2 0.2 0.2 0.2 0.2]
```

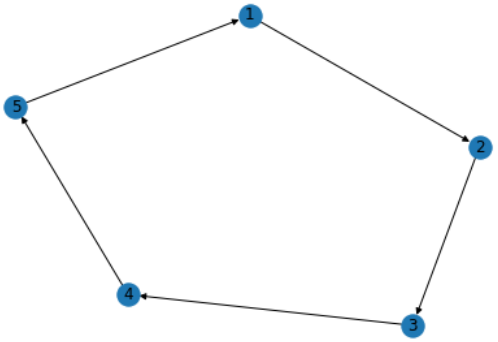
由上圖, 可以看到 graph_2 節點間的關係, 我們可以探討Node 1 的 ageRank 值。由 PageRank algorithm 可以得知 PageRank 它的核心概念是評分越高的 node 通常被許多其他 node 所連結(或reference), 若想增加 Node 1 的

PageRank，可以透過將其餘4 個 node 中其中　個 node 指向 Node 1 (如下圖，node 4指向 node 1)。




```
graph_2.txt
PageRank:
[0.224 0.222 0.219 0.217 0.118]
```

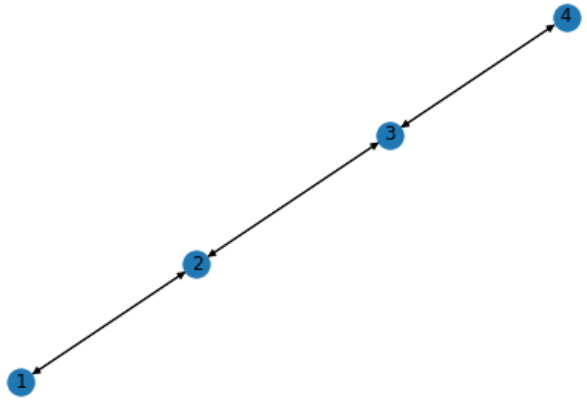
iv. graph_2(Damping Factor = 0.5)




```
graph_2.txt
PageRank:
[0.2 0.2 0.2 0.2 0.2]
```

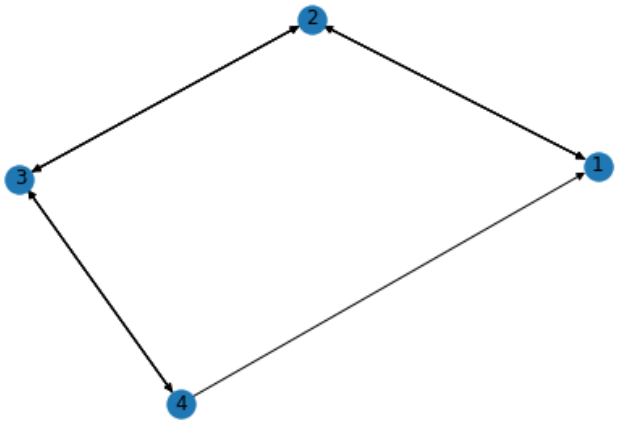
由上圖可以發現更改 Damping Factor 後的 PageRank 完全和更改前的數值一模一樣，由 PageRank 的演算法來看，他們的關聯性的比值是一樣的，因此更改 Damping Factor 對 graph_2 來說是不影響的。

v. graph_3(Damping Factor = 0.1)



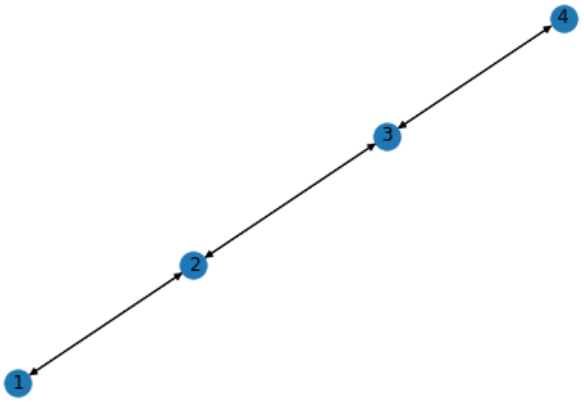

```
graph_3.txt
PageRank:
[0.172 0.328 0.328 0.172]
```

左上圖，可以看到 graph_3 節點間的關係，我們可以探討Node 1 的PageRank 值。由 PageRank algorithm 可以得知 PageRank 它的核心概念是評分越高的 node 通常被許多其他 node 所連結(或reference)，若想增加 Node 1 的 PageRank，可以透過將其餘4 個 node 中其中　個 node 指向 node 1 (如下圖，node 4指向 node 1)。




```
graph_3.txt
PageRank:
[0.25  0.362 0.25  0.138]
```

vi. graph_3(Damping Factor = 0.15)




```
graph_3.txt
PageRank:
[0.175 0.325 0.325 0.175]
```

由上圖可以發現增加了damping factor，PageRank 原本較低的值都變高了；相反的，原本較高的值都變低了。

# 三、SimRank

## 1. 概述:

SimRank 和上述兩種 algorithm 不同, 是　種透過 graphstructure 分析 node 與 node 間相似度的 algorithm, 它的核心概念是若 2 個 node 的 parent 相似, 那麼這 2 個 node 的相似度較高。

可以看到相似度的算法如下圖, 相似度是所有 a node 和b node 的 parents 之組合的相似度總和再平均。

SimRank formula

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

- $I(a)$, $I(b)$: all in-neighbors
- C is decay factot, $0<C<1$
- $S(a, b)\in[0, 1]$
- $S(a, a)=1$

## 2.程式碼簡述:

### i. Node class:

這個class主要是用來建立graph的一個基本單位, 總共有name、children、parents attribute。

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.parents = []
```

### ii. Graph class:

這個class主要包含建立graph一些function, 包含增加邊, 以及尋找Node的function。(這部份的程式碼和HIT、PageRank演算法所使用的方式一模一樣)

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

    def search(self, name):

        exist = False

        for node in self.Nodes:
            if(node.name == name):
                exist = True
                break

        if exist:
            return next(node for node in self.Nodes if node.name == name)

        else:
            new_node = Node(name)
            self.Nodes.append(new_node)
            return new_node

    def addEdge(self, parent, child):
        parent_node = self.search(parent)
        child_node = self.search(child)

        if(child_node.name not in parent_node.children):
            parent_node.children.append(child_node)

        if(parent_node.name not in child_node.parents):
            child_node.parents.append(parent_node)
```

1. addEdge function

和 function 名稱意思相同, 將 parent 和 child 兩個 node連結起來。

2. search function

和 function 名稱意思相同, 用來找尋 graph 中想找到node, 若無該 node, 則建立　個新的 node, 以便後續建立連結。

iii. calSimRank function:

```
def calSimRank(self, node1, node2):
    if(node1.name == node2.name):
        return 1.0

    pNodes1 = node1.parents
    pNodes2 = node2.parents

    if(len(pNodes1) == 0 or len(pNodes2) == 0):
        return 0.0

    SimRankSum = 0
    for node1 in pNodes1:
        for node2 in pNodes2:
            node1Idx = self.nodeList.index(node1.name)
            node2Idx = self.nodeList.index(node2.name)
            SimRankSum += self.oldSim[node1Idx][node2Idx]

    newSimRank = (self.decayFactor / (len(pNodes1) * len(pNodes2))) * SimRankSum
```

由上圖可以將我的程式碼分成
1. 計算 SimRank 的總和
2. 計算 node1,node2 的 SimRank

### iv. SimRank function

```
def SimRank(g, sim, num):
    for i in range(num):
        for node1 in g.Nodes:
            for node2 in g.Nodes:
                rank = sim.calSimRank(node1, node2)
                node1_idx = sim.nodeList.index(node1.name)
                node2_idx = sim.nodeList.index(node2.name)
                sim.newSim[node1_idx][node2_idx] = rank


        sim.oldSim = copy.deepcopy(sim.newSim)
```

這個 function 是透過使用 calSimRank 得到對應 2 個 node 的simRank 值，並將其值建立成　個 SimRank Matrix。

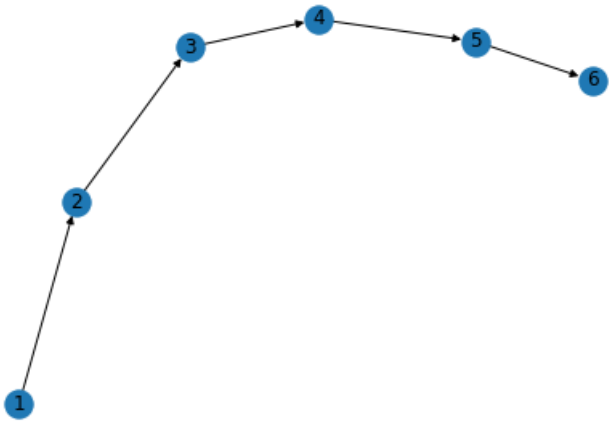
### v. Main

主程式如下, file_path 決定要分析的檔案, it決定iteration的次數, decay_Factor 設定為0.7。其他就是建立Graph以及使用 SimRank function來完成此次任務。

```
if __name__ == '__main__':

    decay_factor = 0.7
    it = 30
    result_dir = 'result'
    data_path = './hw3dataset/'
    for file_name in os.listdir(data_path):
        print(file_name)
        if(file_name == "IBM.txt" or file_name == "graph_6.txt"):
            continue
        file_path = data_path + file_name
        fname = file_path.split('/')[-1].split('.')[0]
        simrank_fname = '_SimRank.txt'

        graph = init_graph(file_path)

        sim = Similarity(graph, decay_factor)
        start = time.time()

        SimRank(graph, sim, iteration)
        end = time.time()
        print(end-start)
        ans = sim.get_sim_matrix()
        print('SimRank:')
        print(ans)
        print()
        path = os.path.join(result_dir, fname)
        os.makedirs(path, exist_ok=True)
        np.savetxt(os.path.join(path, fname + simrank_fname), ans, delimiter=' ', fmt='%.3f')
```

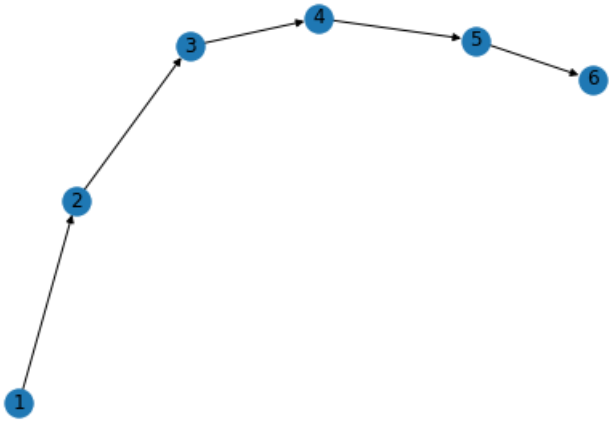
## 3. 結果

### i. graph_1(decay_Factor = 0.7)




```
graph_1.txt
SimRank:
[[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.]]
```

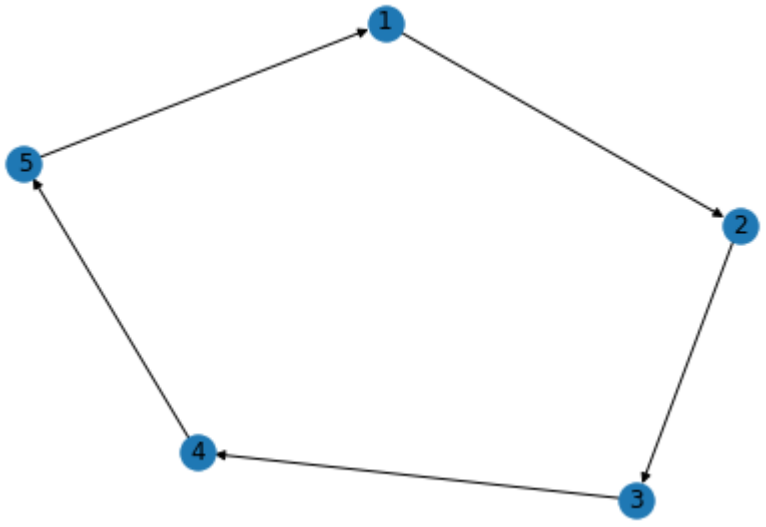
ii. graph_1(decay_Factor = 0.8)




```
graph_1.txt
SimRank:
[[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.]]
```

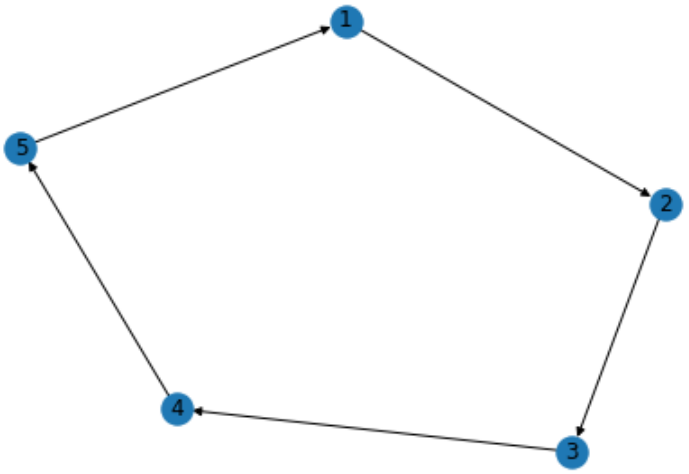
由上圖可以發現變更 decay_Factor 對於 SimRank 是沒有影響的, 因為對於 graph_1 來說，每個 node 彼此之間的相似度都是1，因此更改阻尼係數是無關的。

iii. graph_2(decay_Factor = 0.7)




```
graph_2.txt
SimRank:
[[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.]]
```

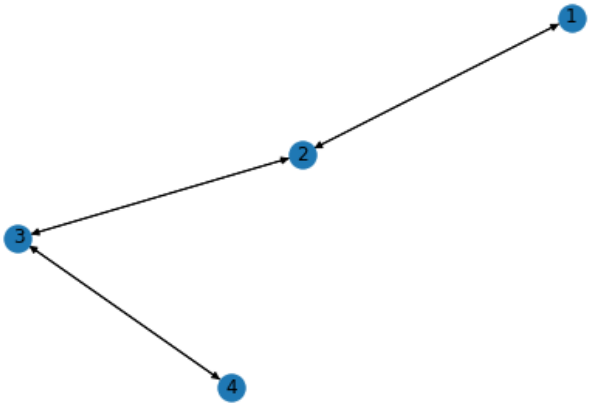
iv. graph_2(decay_Factor = 0.8)




```
graph_2.txt
SimRank:
[[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.]]
```

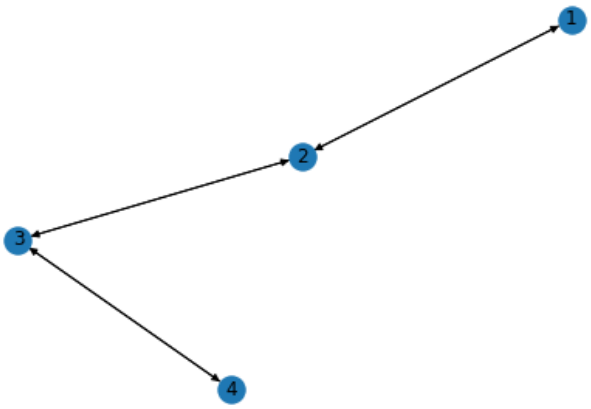
由上圖可以發現變更 decay_Factor 對於 SimRank 是沒有影響的, 因為對於 graph_2 來說，每個 node 彼此之間的相似度都是1，因此更改阻尼係數是無關的。

v. graph_3(decay_Factor = 0.7)




```
graph_3.txt
SimRank:
[[1.    0.    0.538 0.   ]
 [0.    1.    0.    0.538]
 [0.538 0.    1.    0.   ]
 [0.    0.538 0.    1.   ]]
```

vi. graph_3(decay_Factor = 0.8)




```
graph_3.txt
SimRank:
[[1.    0.    0.667 0.   ]
 [0.    1.    0.    0.667]
 [0.667 0.    1.    0.   ]
 [0.    0.667 0.    1.   ]]
```

由上圖顯示, 將 decay_Factor 變大, 除了1以外的 SimRank 都變大了, 可以藉由 SimRank 算法看得出來, 分子變大, 分數就變大

## 四、Effectiveness analysis

1. graph_1
   i.HIT:0.17 ms
   ii.PageRank: 0.1ms
   iii.SimRank:1.03ms
2. graph_2
   i.HIT:0.14 ms
   ii.PageRank: 0.09ms
   iii.SimRank: 0.8ms
3. graph_3
   i.HIT:0.12 ms
   ii.PageRank: 0.07ms
   iii.SimRank:0.6ms
4. graph_4
   i.HIT:0.20 ms
   ii.PageRank: 0.15ms
   iii.SimRank: 10.75ms
5. graph_5
   i.HIT:13.22 ms
   ii.PageRank: 8.91ms
   iii.SimRank: 11787ms
6. graph_6
   i.HIT:49.02 ms
   ii.PageRank: 29.89ms
   iii.SimRank: x
7. IBM
   i.HIT:29.71 ms
   ii.PageRank: 21.65ms
   iii.SimRank: x

這三個演算法中, Page Rank 執行的速度最快, 其次是 HIT 最後是 SimRank 。

1. HIT

其中 HIT 的算法是線性的, 因此所花的時間也會和 Node 數成正比, 若 node 數越多, 花費時間會越來越巨大, 而當 node 數相同時, 鏈結數越多(相同 node 數的狀況下 fullyconnected graph 的鏈結數比單向連通圖多), 花費時間越多, 並且從線性的成長變成非線性成長。

## 2. PageRank

同樣在 PageRank 中, 也會發現和 HITS algorithm 相同的情況, 同樣是比較在相同 node 數量下, 單向連通圖以及 fully-connected graph, 因為鏈結數的不同, 而產生執行效率的不同，但在 PageRank 中可以更明顯地觀察到 fully-connected graph 的成長曲線是非線性的情況

## 3. SimRank

在 SimRank 中, 也會發現和 HITS algorithm 以及 PageRank 相同的情況, 同樣是比較在相同 node 數量下, 單向連通圖以及 fully-connected graph, 因為鏈結數的不同, 而產生執行效率的不同，但在 SimRank 中可以直接地觀察到 fully-connected graph 的成長曲線是非線性的情況, 比起 HITS 和 PageRank 而言更加明顯。
這是因為 SimRank 的算法為

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

可以看到紅色框起來的部分, 若應用在 fully-connected graph 時, 若有 n 個 node, 則這邊就必須計算 n 的平方個 SimRank 值，因此在 fully-connected graph 下，當 node 越多, 所花費的時間就越多。