Data Mining HW3

NE6101034

AI 碩一 柳譯筑

Implementation detail

我的檔案使用方式：terminal 輸入以下指令，graph_[num] // num為第幾張圖的羅馬數字

另外使用 ibm dataset 則是隨便輸入

```
python3 DM_hw3.py [graph_1]
```

Dataset

- 我把 graph data 放在 hw3dataset 這個資料夾裡面
- About ibm-dataset:

  作業一的 ibm 檔案是給 transaction id 跟 itemset , 我先處理成 dictionary 再做 permutation，（才能弄成雙向graph)，製作出 bidirectional graph

```
# for graph 1-5
def make_graph(lines):
    graph = Graph()
    for line in lines:
        [parent, child] = line.strip().split(',')
        graph.add_edge(parent, child)
    graph.sort_nodes
    return graph

# for ibm dataset
def make_graph_ibm(ibm_dict):
    graph = Graph()
    lst = []
    for itemset in ibm_dict.items():
        lst.append(list(permutations(itemset[1],2)))
    for i in range(len(lst)):
        for j in lst[i]:
            [parent, child] = list(j)
            graph.add_edge(parent,child)
        graph.sort_nodes
    return graph
```

Algorithm

我將這個 project 分為兩個檔

第一個是主要執行的檔案，我命名為DM_hw3.py

第二個是放這個檔案所需要的一些 class，像是 graph 跟 node 皆為會用到的 class，我命名為DM_src.py

首先介紹 src 裏面各個演算法都會需要用到的 graph class 跟 node class

- class Graph
  - 關於一些Graph的基本function

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

```
        self.num = 0

  # 這個函式負責找是否存在這個node
    def contains(self, name):
        for node in self.nodes:
            if(node.name == name):
                return True
        return False

    # 如果找到會回傳這個node的name, 如果沒有找到則會創建一個新的node並且回傳新的node name
    def find(self, name):
        f(not self.contains(name)):
            new_node = Node(name)
            self.nodes.append(new_node)
            return new_node
        else:
            return next(node for node in self.nodes if node.name == name)
    # 將新的子節點跟父節點連結
    def add_edge(self, parent, child):
        parent_node = self.find(parent)
        child_node = self.find(child)

        parent_node.link_child(child_node)
        child_node.link_parent(parent_node)

    # 在 make graph 的時候會叫到這個function去排序graph裏面的nodes
    def sort_nodes(self):
        self.nodes.sort(key = lambda x : int(x.name))
```

- class Node

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

    # 新的edge產生 parent要去連 child node
    def link_child(self, new_child):
        for child in self.children:
            if(child.name == new_child.name):
                return None
        self.children.append(new_child)

    # 新的edge產生 child node 要連 parent node
    def link_parent(self, new_parent):
        for parent in self.parents:
            if(parent.name == new_parent.name):
                return None
        self.parents.append(new_parent)
```

HITS

主要在實作這個演算法：

- Recursive dependency:

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

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

- Using Linear Algebra, we can prove:

a(v) and h(v) converge

HubsAuthorities(G)

$$\mathbf{1} \leftarrow [1, \ldots, 1] \in R^{|V|}$$

$$a_0 \leftarrow h_0 \leftarrow \mathbf{1}$$

$$t \leftarrow 1$$

repeat

for each v in V

$$\text{do } a_t(v) \leftarrow \sum_{w \in pa[v]} h_{t-1}(w)$$

$$h_t(v) \leftarrow \sum_{w \in ch[v]} a_{t-1}(w)$$

$$\boxed{\begin{aligned} a_t &\leftarrow a_t / \| a_t \| \\ h_t &\leftarrow h_t / \| h_t \| \\ t &\leftarrow t + 1 \end{aligned}}$$

normalization

$$\text{until } \| a_t - a_{t-1} \| + \| h_t - h_{t-1} \| < \varepsilon$$

$$\text{return } (a_t, h_t)$$

- 這是一些放在 graph class 裏面的 function

```
# 存normalized 的 auth 跟 hub
# at <- at/||at||
# ht <- ht/||ht||
def normalize_auth_hub(self):
        auth_sum = sum(node.auth for node in self.nodes)
        hub_sum = sum(node.hub for node in self.nodes)

        for node in self.nodes:
            node.auth /= auth_sum
            node.hub /= hub_sum

# 這個function 存 hits 的 hub 跟 authority
def save_hub_auth(self):
  lst_auth = [node.auth for node in self.nodes]
  lst_hub = [node.hub for node in self.nodes]
  np.savetxt('graph_'+ str(self.num)+'_HITS_authority'  + '.txt',np.asarray(lst_auth),fmt='% 1.5f',delimiter=' ',newline='' )
  np.savetxt('graph_'+str(self.num)+'_HITS_hub_' + '.txt',np.asarray(lst_hub),fmt='% 1.5f',delimiter=' ',newline = '')
```

- 這是一些放在 node class 裏面的 function

```
#sum of parents' hub
def update_auth(self):
```

```
    self.auth = sum(node.hub for node in self.parents)
# sum of children's hub
def update_hub(self):
    self.hub = sum(node.auth for node in self.children)
```

- HITS主要演算法：

```
# 每做一回合的 hits 呼叫一次這個 function
def HITS_one_iter(graph):
        node_list = graph.nodes
        # each node 會 update 一次 authority
        for node in node_list:
            node.update_auth()
        # each node 會 update 一次 hub
        for node in node_list:
            node.update_hub()

        graph.normalize_auth_hub()

def HITS(graph, iter):
    for i in range(iter):
        HITS_one_iter(graph)
    graph.display_hub_auth()
    return graph

# 在 main 執行
HITS(graph,iter)
```

Page Rank

主要在實作這條式子：

- 我將 damping factor 設為 0.15

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

D(damping factor)=0.1~0.15

n=|page set|

- 這是一些放在 graph class 的 function

```
# 這個function會回傳一個 pagerank list 的np array
def get_pagerank_list(self):
    pagerank_list = np.asarray([node.pagerank for node in self.nodes], dtype='float32')
    return np.round(pagerank_list, 3)

# 這個 function 儲存 pagerank
def save_pagerank(self):
    lst = [node.pagerank for node in self.nodes]
    np.savetxt('graph_'+ str(self.num)+'_PageRank' + str(self.num) + '.txt',np.asarray(lst),fmt='% 1.5f',delimiter=' ',newline = '')
```

- 這是一些放在 node class 的 function

```
# normalize pagerank list 的數值
def normalize_pagerank(self):
    pagerank_sum = sum(node.pagerank for node in self.nodes)

    for node in self.nodes:
      node.pagerank /= pagerank_sum

# 實作page rank 的公式
def update_pagerank(self, d, n):
      # in_nodes 為此node所有父節點的集合
      in_nodes = self.parents
      # sum pagerank(ni) / C(ni)
      pagerank_sum = sum((node.pagerank / len(node.children)) for node in in_nodes)
      random_jumping = d / n
      self.pagerank = random_jumping + (1-d) * pagerank_sum
      #另外有一個算式是 pagerank = d + (1-d) * pagerank_sum
```

- page rank 主要執行演算法：

```
# 主要呼叫function
def PageRank(graph,damping_factor,iter):
    for i in range(iter):
        PageRank_one_iter(graph, damping_factor)
    graph.display_pagerank_list()

# 每呼叫一次 update 一個 iteration
def PageRank_one_iter(graph, d):
    node_list = graph.nodes
    for node in node_list:
        node.update_pagerank(d, len(graph.nodes))
    graph.normalize_pagerank()

# 在 main 執行
iter = 100
damping_factor = 0.15
PageRank(graph, damping_factor, iter)
```

Sim Rank

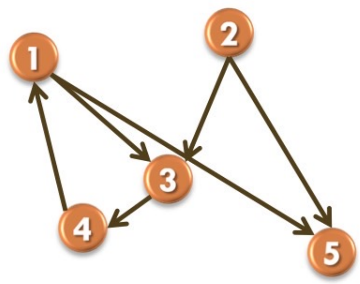
- 我將 decay factor C 設為 0.9

主要在實作這條式子：

SimRank formula

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

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



1'st iteration
S(3, 5)=C/4 * 2
S(4, 5)=0

How about S(4,5) while e(1,2) is added?

- 用 SimRank 這個 class 放一些 simRank 的計算
- [note] 我把演算法流程註解在以下程式碼中

```
# class SimRank 內
def __init__(self, graph, decay_factor):
    self.decay_factor = decay_factor
    self.name_list, self.old_sim = self.init_sim(graph)
    # name_list 是所有nodeㄉ名字 ，一開始 initialize old_sim
    self.node_num = len(self.name_list)
    # 計算總共幾個node
    self.new_sim = [[0] * self.node_num for i in range(self.node_num)]
    # initialize new simrank 的 list
    self.num = graph.num

    # initialize similarity
    def init_sim(self, graph):
        nodes = graph.nodes
        name_list = [node.name for node in nodes]
        sim = []
        for name1 in name_list:
            temp_sim = []
            #(ex node1 -> temp_sim.append[1,0,0,0,0,0])
            for name2 in name_list:
                if(name1 == name2):
                    temp_sim.append(1)
                else:
                    temp_sim.append(0)
            sim.append(temp_sim)
            # sim is a 2d-array of similarity

        return name_list, sim

    # return the index of node X in node array(name_list)
    def get_name_index(self, name):
        return (self.name_list.index(name))

    # return S(a,b)
    def get_sim_value(self, node1, node2):
```

```
        node1_idx = self.get_name_index(node1.name)
        node2_idx = self.get_name_index(node2.name)
        return self.old_sim[node1_idx][node2_idx]

    # update to new similarity
    def update_sim_value(self, node1, node2, value):
        node1_idx = self.get_name_index(node1.name)
        node2_idx = self.get_name_index(node2.name)
        self.new_sim[node1_idx][node2_idx] = value

    # update old similarity of new similarity now
    def replace_sim(self):
        for i in range(len(self.new_sim)):
            self.old_sim[i] = self.new_sim[i]

    def calculate_SimRank(self, node1, node2):
        # Return 1 if they are same node
        if(node1.name == node2.name):
            return 1.0

        # Use 2 arrays to save 2 nodes' parents
        in_neighbors1 = node1.parents
        in_neighbors2 = node2.parents

        # Return 0 if one of them has no in-neighbor
        if(len(in_neighbors1) == 0 or len(in_neighbors2) == 0):
            return 0.0

        SimRank_sum = 0
        # 用雙迴圈去計算兩者的 sum(parents 的 old_sim)
        for in1 in in_neighbors1:
            for in2 in in_neighbors2:
                SimRank_sum += self.get_sim_value(in1, in2)

        # Implement the equation S(a,b) = (C / |I(a)||I(b)|) * sum of S(Ii(a),Ii(b))
        scale = self.decay_factor / (len(in_neighbors1) * len(in_neighbors2))
        new_SimRank = scale * SimRank_sum

        return new_SimRank
```

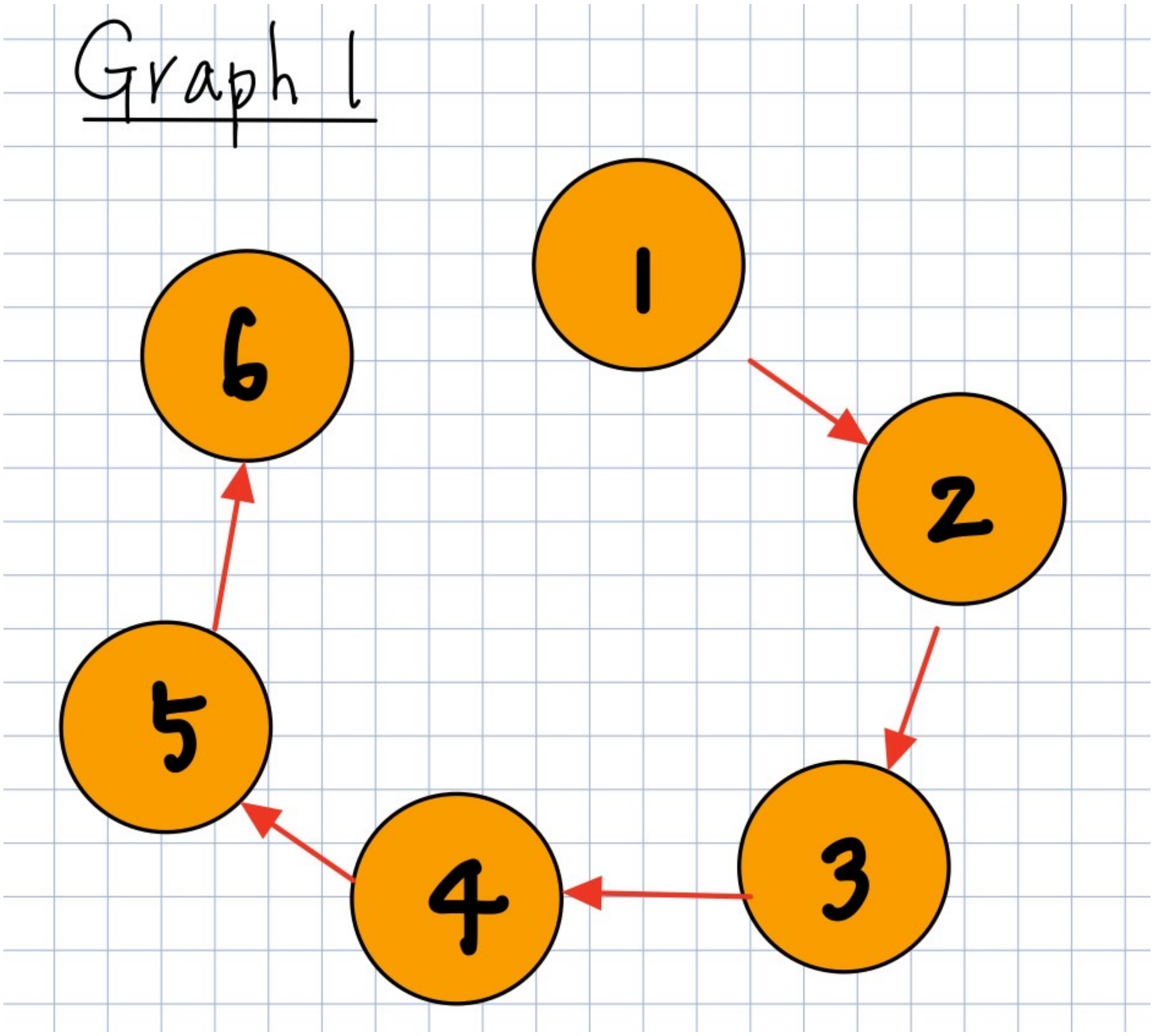
- SimRank 主要執行演算法：

```
# 每呼叫一次更新一次 node1 node2 similarity
def SimRank_one_iter(graph, sim):
    for node1 in graph.nodes:
        for node2 in graph.nodes:
            next_SimRank = sim.calculate_SimRank(node1, node2)
            sim.update_sim_value(node1, node2, next_SimRank)
# 主要呼叫simrank function
def SimRank(graph, sim, iteration=100):
    for i in range(iteration):
        SimRank_one_iter(graph, sim)
  sim.replace_sim()

# 設定sim rank的參數
decay_factor = 0.9
iter = 100
# 執行SimRank
sim = SimRank(graph,decay_factor)
start = time.time()
SimRank(graph,sim,iter)
print("Total Time of SimRank: ", time.time()-start, " sec")
```

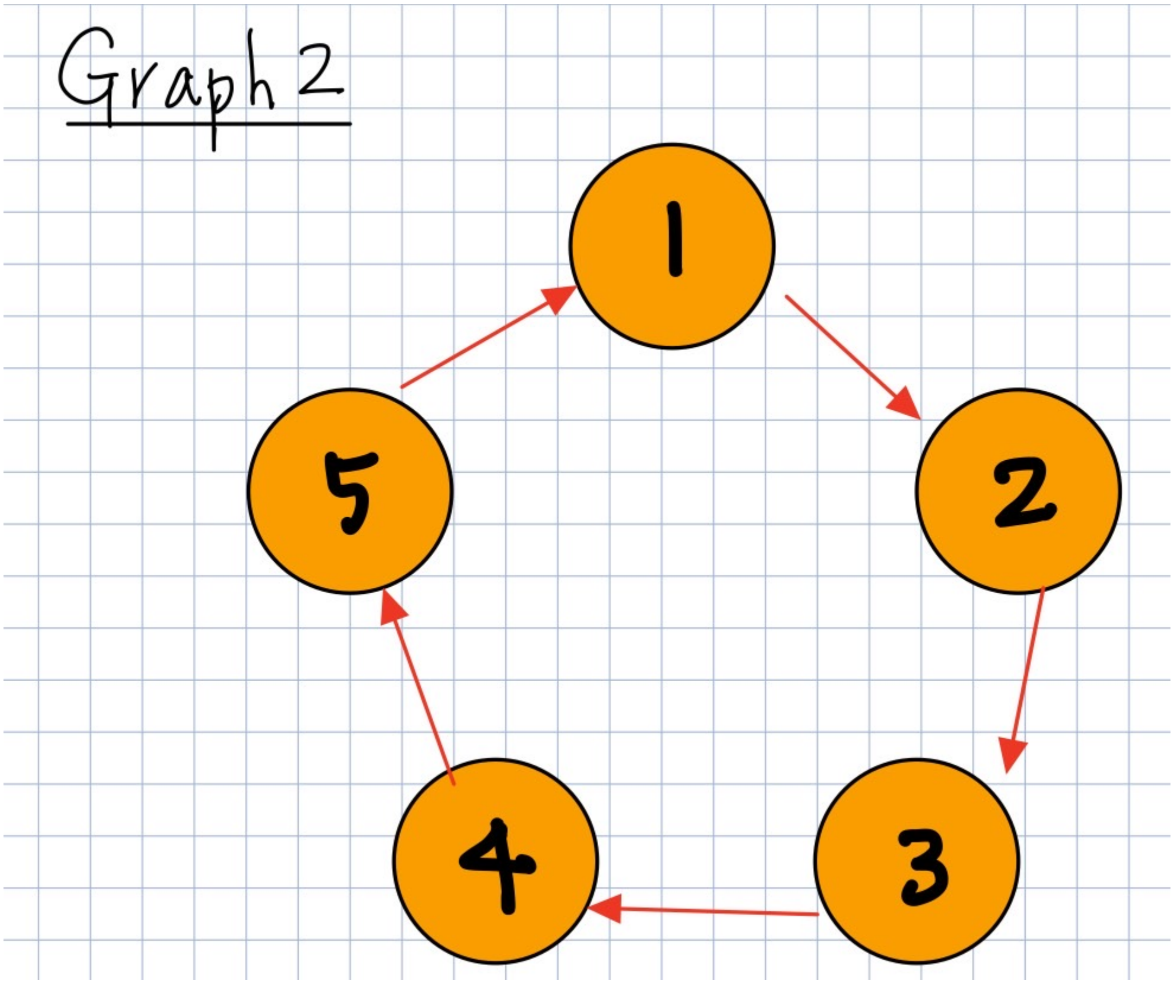
Result analysis and discussion

- Graph
    - graph1



  - hub = [0.20000 0.20000 0.20000 0.20000 0.20000 0.00000]
  - authority = [0.00000 0.20000 0.20000 0.20000 0.20000 0.20000]
  - pagerank = [0.06072 0.11232 0.15619 0.19348 0.22517 0.25211]
- 分析結果：
  - 只有 node1 沒有被任何node指到，所以 node1的 authority 是0
  - 只有 node6 沒有指到任何node，所以 node6 的 hub 是0
  - 由於他是一條 link 起來的 graph, 所以他的page rank是疊加上去的

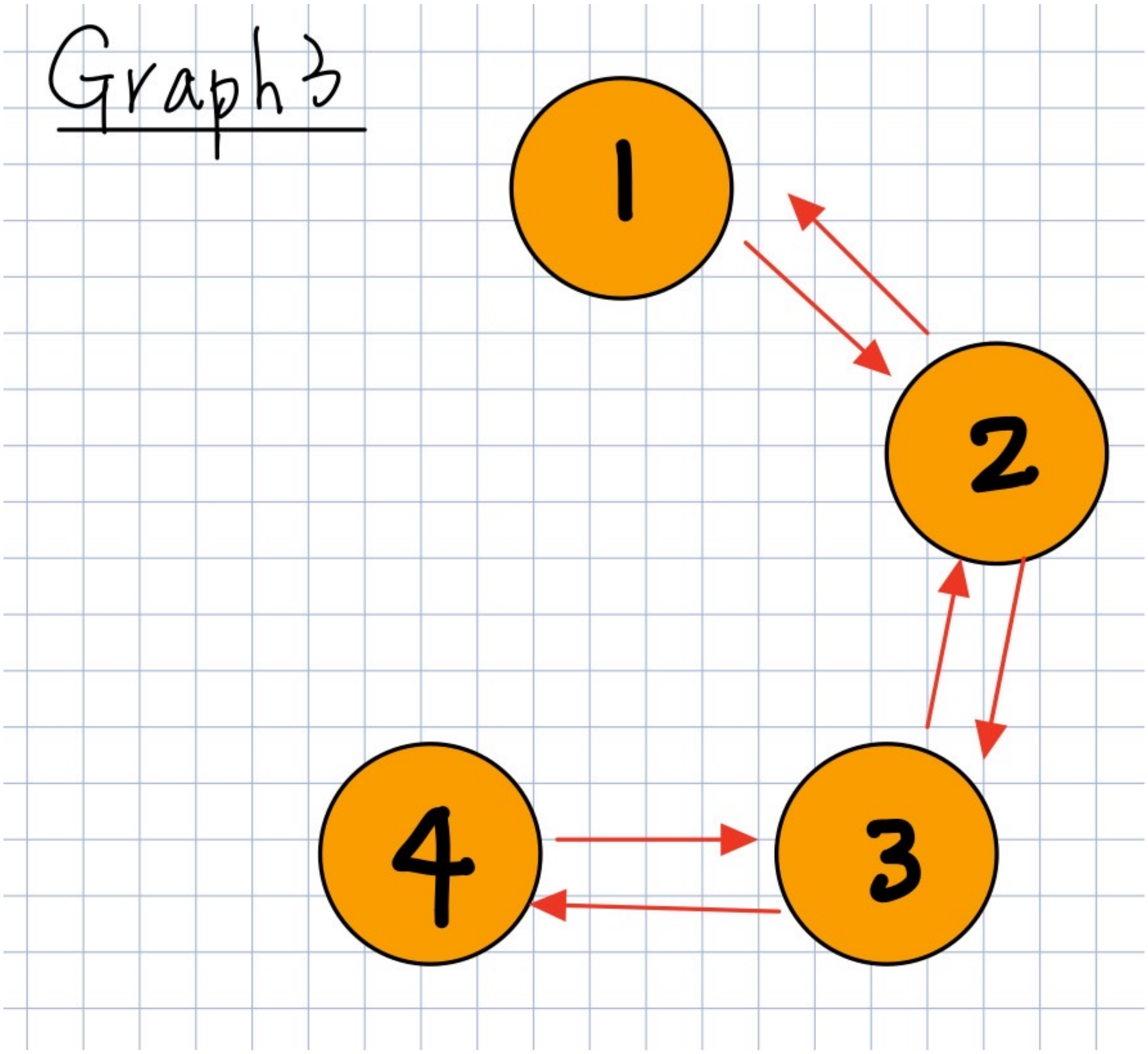
- graph2



- hub = [0.20000 0.20000 0.20000 0.20000 0.20000]
- authority = [0.20000 0.20000 0.20000 0.20000 0.20000]
- pagerank = [0.20000 0.20000 0.20000 0.20000 0.20000]

- 分析結果：
  - 每一個 node 都是 out deg =1 跟一個 in deg = 1 , 所以每個 nodes 的 hub 皆為一樣
  - pagerank 也是同理 他是一個 circular graph 所以考量只到父節點的 page rank，每個node 的 page rank是一樣的

- graph3



- hub = [0.19098 0.30902 0.30902 0.19098]
- authority = [0.19098 0.30902 0.30902 0.19098]
- pagerank = [0.17544 0.32456 0.32456 0.17544]

- 分析結果：

  node1 跟 node4 都是 in deg =1   out deg =1

  node2 跟 node3 都是 in deg=2, out deg=2, 所以算出來 hub 跟 authority 是一樣的

  而 pagerank 這個演算法會考慮到所有父節點的 page rank, 所以得到的結果又會比 HITS在高一點點

- 針對不同的 damping factor 與 decay factor:
  - 根據講義的建議，damping factor 的 range 建議在 (0.10 , 0.15)
    - 我拿 graph 5 來實測
      - damping factor = 0.30

- [0.257146349979801, 0.1530014577560855, 0.1375098525496496, 0.11149447504025642, 0.18649624677680815, 0.07885763185431499, 0.07549398604308429]

- damping factor = 0.20
  - [0.27257372197170443, 0.156667131545101, 0.13837881394015467, 0.10924643205801685, 0.1853160398555778, 0.07218322408690128, 0.06563463654254413]
- damping factor = 0.15
  - [0.2802877979895022, 0.15876448951901675, 0.1388818183465401 2, 0.1082195987115897, 0.1841981252931901, 0.06907749708678682, 0.060570673053374324]
- damping factor = 0.10
  - [0.28801190350363043, 0.16104084511965652, 0.13942020866019816, 0.10724631435399859, 0.18274869972280364, 0.06612785691636776, 0.05540417172334511]

- decay factor
  - 拿 graph5，做 10 個 iteration
    - decay factor = 0.9
      - Total Time of SimRank : 58.30062222480774 sec
      - [1. 0. 0. ... 0. 0. 0.]
        [0. 1. 0.9 ... 0. 0. 0.]
        [0. 0.9 1. ... 0. 0. 0.]
        ...
        [0. 0. 0. ... 1. 0.9 0.9]
        [0. 0. 0. ... 0.9 1. 0.9]
        [0. 0. 0. ... 0.9 0.9 1.]]
    - decay factor = 0.8
      - [[1. 0. 0. ... 0. 0. 0.]
        [0. 1. 0.8 ... 0. 0. 0.]
        [0. 0.8 1. ... 0. 0. 0.]
        ...
        [0. 0. 0. ... 1. 0.8 0.8]
        [0. 0. 0. ... 0.8 1. 0.8]
        [0. 0. 0. ... 0.8 0.8 1.]]
  - graph 4
    - decay = 0.9
      - [[1. 0.55424036 0.5442225 0.54701471 0.53435082 0.49898906 0.59504036]
        [0.55466411 1. 0.58876555 0.55848974 0.59637582 0.62520385 0.49177562]
        [0.54506322 0.5893886 1. 0.62143514 0.5769692 0.62286512 0.62000517]
        [0.54827339 0.55974484 0.62212852 1. 0.53714786 0.6895958 0.69045787]
        [0.53669766 0.5979065 0.57802517 0.53802724 1. 0.59404361 0.48201087]
        [0.50126716 0.62737116 0.62440966 0.69045787 0.59423493 1.

0.48091574]
[0.59803707 0.49582502 0.62241747 0.69151395 0.48392536 0.48302789

]]

- decay = 0.8
  - [[1.   0.35930577 0.34796472 0.35276328 0.336597   0.29128516
    0.4142414]
    [0.35937148 1.   0.40592532 0.368767   0.41136774 0.45320979
    0.28432421]
    [0.34809669 0.40602197 1.   0.44882669 0.38926815 0.45027397
    0.4473794]
    [0.35295615 0.36895584 0.44892876 1.   0.3418527 0.53451581
    0.5346388]
    [0.33694656 0.41159561 0.38942006 0.34197749 1.   0.4116315
    0.27232348]
    [0.29162178 0.45352747 0.45049607 0.5346388 0.41165797 1.
    0.2692776]
    [0.41466783 0.28489176 0.44771308 0.53477863 0.27257581 0.26955725
    1.]]
- 分析結果： decay factor 單純影響 similarity 0 跟 1 之外的值的 scale

關於時間複雜度：

HITS:

HITS 每個 iteration 會去 update iterate 整個 node list 的auth 跟 hub

```
# for each node
self.auth = sum(node.hub for node in self.parents)
# for each node
self.hub = sum(node.auth for node in self.children)
```

時間複雜度來看是約 O(k*N*E)的複雜度, k為iteration, N 為 node, E 為 edge

PageRank:

PageRank 每個 iteration 會去 iterate 整個 node list

整個 node list 的 node 又會去iterate in nodes

```
pagerank_sum = sum((node.pagerank / len(node.children)) for node in in_nodes)
```

約 O(k*N*E)的複雜度 k為iteration, N為 node , E 為 edge

SimRank:

從演算法來看，simRank 每個 iteration 會計算 node1(from node_list), node2(from node list)的 similarity

```
for node1 in graph.nodes:
  for node2 in graph.nodes:
    new_SimRank = sim.calculate_SimRank(node1, node2)
    sim.update_sim_value(node1, node2, new_SimRank)
```

計算 similarity 又需要去迭代 node1 跟 node2 的 parents

```
for in1 in in_neighbors1:
  for in2 in in_neighbors2:
    SimRank_sum += self.get_sim_value(in1, in2)
```

SimRank 的 Space complexity is O(n^2) , time complexity is *O(k**n^2*d) :

k is the number of iterations

n is the number of node, d is the average of product of in degrees of pair of vertices.

Effectiveness analysis

Computation performance analysis

以下是iterator設定為 10

- 我測試跑比較大的圖來比較不同演算法之間的時間差異

  graph_1.txt: 6 nodes, 5 edges
  graph_2.txt: 5 nodes, 5 edges (a circle)
  graph_3.txt: 4 nodes, 6 edges
  graph_4.txt: 7 nodes, 18 edges (the example in Lecture3, p29)
  graph_5.txt: 469 nodes, 1102 edges
  graph_6.txt: 1228 nodes, 5220 edges

- Graph 4
  - Total Time of HITS: 0.0013761520385742188 sec
  - Total Time of pagerank: 0.0006701946258544922 sec
  - Total Time of SimRank: 0.01285696029663086 sec
- Graph 5
  - Total Time of HITS: 0.008016824722290039 sec
  - Total Time of pagerank: 0.0036149024963378906 sec
  - Total Time of SimRank: 58.628607988357544 sec
- Graph 6
  - Total Time of HITS: 0.019309043884277344 sec
  - Total Time of pagerank: 0.011114358901977539 sec
  - Total Time of SimRank: 222.20421504974365 sec (這邊只跑了一個iterate 因為太久了)
- IBM_DataSet
  - Total Time of HITS: 0.04976081848144531 sec
  - Total Time of pagerank: 0.04711174964904785 sec

以下是 iterator 設定為 100 只跑了graph4 graph5 (其他的要算非常久...)

- Graph 4
    - Total Time of HITS: 0.0527949333190918 sec
    - Total Time of pagerank: 0.02937793731689453 sec
    - Total Time of SimRank: 58.47192907333374 sec

- Graph5
    - Total Time of HITS: 0.04401206970214844 sec
    - Total Time of pagerank: 0.029086828231811523 sec
    - Total Time of SimRank: 599.5495269298553 sec
- Graph6
    - Total Time of HITS: 0.1290278434753418 sec
    - Total Time of pagerank: 0.09269189834594727 sec

我發現針對不同演算法，node 的數量會隨著複雜度越複雜，影響整體跑的時間就越久

到 graph 6 的 node 數到 1228，edge 有 5220條，SimRank 只跑一個 iterator要跑超級久（3~4分鐘）

Discussion and Experience

HITS

- 對網頁進行評級
- 與查詢相關，Authority 跟 hub 分數受搜索詞的影響
- 一個好的中心代表一個指向許多其他頁面的頁面
- 一個好的權威機構代表一個由許多不同的中心連結的頁面
- Authority 跟 hub 在互相遞歸中根據彼此定義
- 他跟 pagerank 與 SimRank 比起來較少使用在搜尋引擎上

PageRank

- PageRank 的基本假設是：更重要的頁面往往更多被其他頁面參照(refernce)

- 可以應用於任何含有元素之間相互參照的情況的Set
- 將網頁視為node, 連結視為edge
- 每個網頁的權重值大小被遞迴地定義，依託於所有連結該頁面的頁面的權重值
- PageRank可以想像為vote，每人只能投一票，如果投給兩個人那就是各1/2票
- PageRank 只能衡量每個結點的重要性，SimRank 相似度能比較任意兩個結點間的相似度問題

SimRank

- SimRank 的基本假設是：自己跟自己最相關

- SimRank 的計算需要 O(k|E|^2) 的時間跟 O(|V|^2)的空間，透過需要很多時間的遞迴把自己跟其他點做關聯，在信息檢索領域像是搜尋引擎優化、協同過濾推薦、文檔聚合分類都很符合這個理論。
- 與傳統的文本相似度（Textual Similarity）相比，SimRank 相似度的計算完全基於網絡圖的拓撲結構，他遞歸的定義方式使SimRank相似度的值捕捉到圖結構的整體。

- 感謝助教看完這份很長的 report～新年快樂～～