Abstract

實作三種 Link Analysis 演算法並分析資料。

1. Related Work

所有程式皆在 colab 環境執行，利用 pandas 讀取 txt 檔。使用 networkx 函式庫，根據讀取值以及 DiGraph() 和 add_edge(u,v)建立有向圖 G。其中透過 networkx 函式庫建立的圖 G 能輕鬆繪製圖以其訪問其父節點與子節點。

1.1. HITS Algorithm description

透過 time 函式庫計算演算法執行時間。建立 node 類別進行參數初始化以及 authority 和 hub 的處存，核心演算法為迭代 k 次來分別更新 authority 和 hub。對圖 G 中所有頂點 p 遍歷其父節點 q 計算 authority 以及 norm，最後再將計算出的 authority 除上 norm。hub 的計算則是對圖 G 中所有頂點 p 遍歷其父節點 q 計算 hub 以及 norm，最後再將計算出的 hub 除上 norm。最後計算出 authority 和 hub。

```
def HITS(G, k: int = 100):
  start = time.time()

  # 建立類別
  class node():
    def __init__(self):
      self.auth = []
      self.hub = []
    def set_initial(self, nodenumber):
      self.auth = np.ones(nodenumber+1)
      self.hub = np.ones(nodenumber+1)

  # 初始化
  node = node()
  node.set_initial(max(G.nodes))

  # 迭代
  for i in range(1,k+1,1):
  ### 更新 auth
    norm = 0
    for p in G.nodes:
      node.auth[p] = 0
      for q in G.predecessors(p):
        node.auth[p] += node.hub[q]
      norm += node.auth[p]**2
    norm = sqrt(norm)
    for p in range(len(G.nodes)):
      node.auth[p] = node.auth[p] / norm
  ### 更新 hub
    norm = 0
    for p in G.nodes:
      node.hub[p] = 0
      for r in G.neighbors(p):
        node.hub[p] += node.auth[r]
      norm += node.hub[p]**2
    norm = sqrt(norm)
    for p in range(len(G.nodes)):
      node.hub[p] = node.hub[p] / norm
  end = time.time()
  print("執行時間：%f 秒" % (end - start))

  return node.auth[1:], node.hub[1:]
```

1.2. PageRank Algorithm description

透過 time 函式庫計算演算法執行時間。並使用 networkx 函式庫將圖 G 轉換成鄰接矩陣 G_m，因為矩陣相乘算符@是定義在 ndarray 的資料形態下，因此透過 np 函式庫的函式將 G_m 轉換成資料型態為 ndarray 的矩陣 M，最後透過 k 次的迭代計算出 pagerank v。

```
def pagerank(G, k: int = 100, d: float = 0.85)
:
  start = time.time()

  G_m = nx.to_numpy_matrix(G)
  M = np.squeeze(np.asarray(G_m))

  N = M.shape[1]
  v = np.ones(N) / N
  M_hat = (d * M + (1 - d) / N)
  for i in range(k):
      v = M_hat @ v

  end = time.time()
  print("執行時間：%f 秒" % (end - start))

  return v
```

1.3. SimRank Algorithm description

原始的 SimRank 計算方式是透過不斷迭代計算該點父節點的 SimRank，因此如果遇到迴圈或是複雜的圖，其計算量會十分龐大，最壞情況下其時間複雜度為 O(n^n)，因此將 SimRank 改為迭代 k 次的近似是十分必要的。

透過新、舊 sim 處存 SimRank 的值，每次迭代都將新 sim 的值賦予舊 sim，接著將新計算出的 sim 值賦予新 sim。初始化為建立單位矩陣。

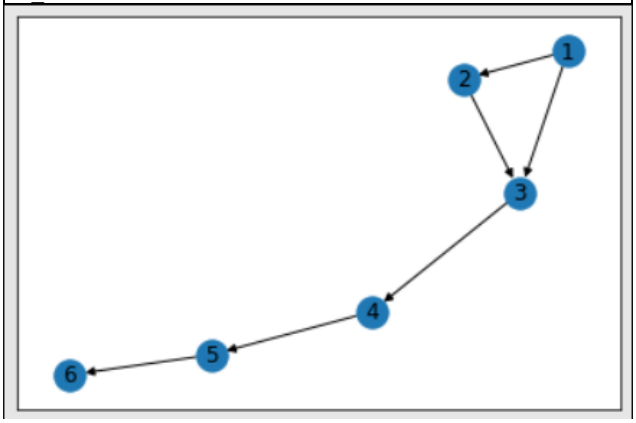
每次迭代都重新計算圖 G 中的每一頂點與彼此的 SimRank 值。若此時兩頂點分別為 u 和 v，u 等於 v 時回傳 1；u 和 v 其中一點的父節點數為零時則回傳 0；非以上兩種情況則遍歷 u 父節點 w 以及 v 父節點 x，透過公式與舊 sim，計算出 u 對應 v 的 SimRank。

```
### 計算
def calculate_SimRank(G, sim, u, v, d):
  if(u==v):
    return 1

  in_neighbors_u = list(i for i in G.predecessors(u))
  in_neighbors_v = list(i for i in G.predecessors(v))

  if(len(in_neighbors_u) == 0 or len(in_neighbors_v) == 0):
    return 0

  SimRank_sum = 0
  for w in in_neighbors_u:
    for x in in_neighbors_v:
      SimRank_sum += sim[w,x]

  scale = d / (len(in_neighbors_u) * len(in_neighbors_v))
  new_SimRank = scale * SimRank_sum

  return new_SimRank

### 迭代
def SimRank_one_iter(G, sim, d):
  new_SimRank = sim
  for u in G.nodes:
    for v in G.nodes:
      new_SimRank[u,v] = calculate_SimRank(G, sim, u, v, d)
  return new_SimRank

### 初始化
def init_sim(G):
  sim = np.zeros((max(G.nodes)+1, max(G.nodes)+1))
  for i in range(len(sim)):
    sim[i,i] = 1
  return sim

### 執行
def get_simrank(G,d,k):
  start = time.time()

  old sim = init sim(G)
  new sim = old sim

  for i in range(k):
    old sim = new sim
    new sim = SimRank one iter(G, new sim, d)

  end = time.time()
  print("執行時間：%f 秒" % (end - start))

  return new_sim[1:,1:]
```

2. Find a way to increase hub, authority, and PageRank of Node 1

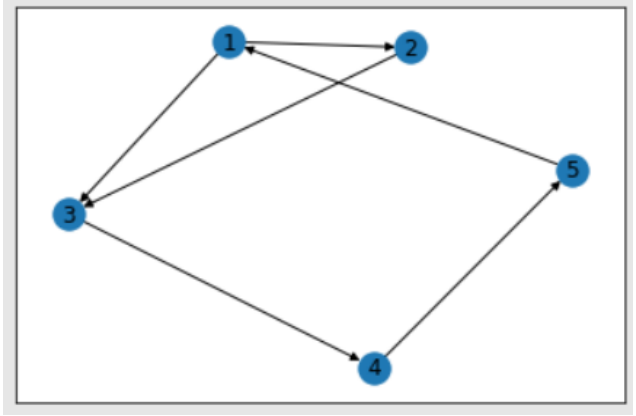
針對 graph_1.txt 所建立的圖 G1，若增加有向邊 e(1,3)，可提高頂點 1 authority, hub 和 PageRank 三項數值。因為增加了頂點 1 的子節點同時增加了頂點 1 的分支度。

```
auth： 0.3333333333333333
auth_a： 0.5257311121190577
hub： 0.3333333333333333
hub_a： 0.5257311121190577
v： 0.1016376038415808
v_a： 0.15905858584641847
```



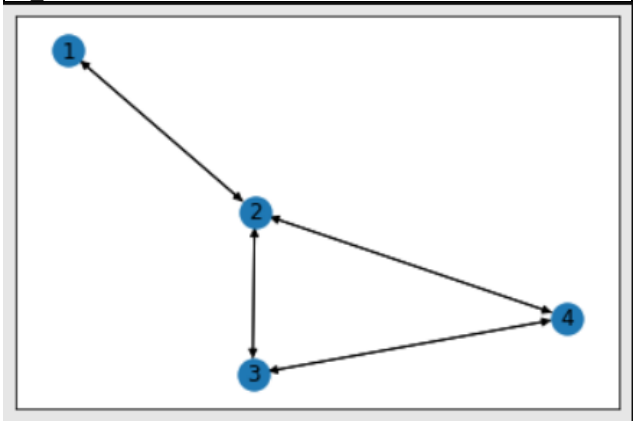
針對 graph_2.txt 所建立的圖 G2，若增加有向邊 e(1,3)，可提高頂點 1 authority, hub 和 PageRank 三項數值。因為 G2 為循環圖，增加了頂點 1 的子節點同時增加了頂點 1 的分支度，因此可以增加頂點 1 的三項數值。

```
auth： 0.31034298554859197
auth_a： 0.5257311121190159
hub： 0.31034298554859197
hub_a： 0.5257311121190397
v： 0.19999999999999998
v_a： 0.3503252008608161
```



針對 graph_3.txt 所建立的圖 G3，若增加有向邊 e(2,4)和有向邊 e(4,2)，可提高頂點 1 authority, hub 和 PageRank 三項數值。因為 G3 為的邊都是雙向邊，雖然沒有增加頂點 1 的父節點數、子節點數以及分支度，但增加了其他的頂點的三項數值的同時亦可以提升頂點 1 的重要程度，尤其是在提升頂點 2 的重要程度後。

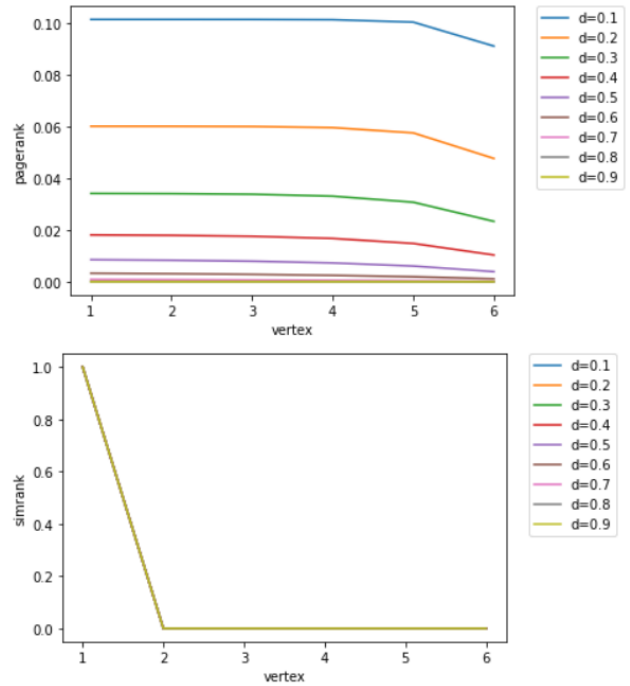
```
auth： 0.5537647602692684
auth_a： 0.6560218873976725
hub： 0.5405353104935973
hub_a： 0.6560218873976725
v： 1.2025555525072718
v_a： 5.256995681404565
```



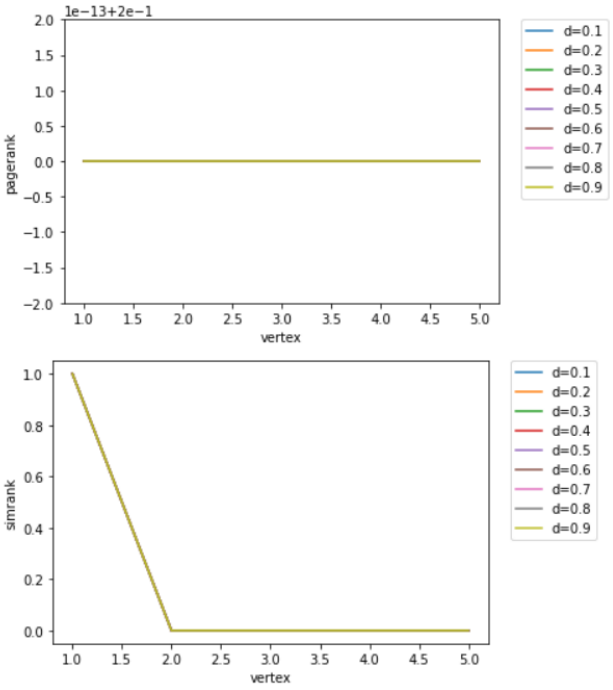
3. Effect with damping factor and decay factor

以下三張圖分別表示 damping factor 和 decay factor 的變化對圖 G1、G2、G3 的影響，上圖表示每個點在不同 damping factor 下的 pagerank，下圖表示不同 decay factor 頂點 1 對其他幾個點的 simrank。

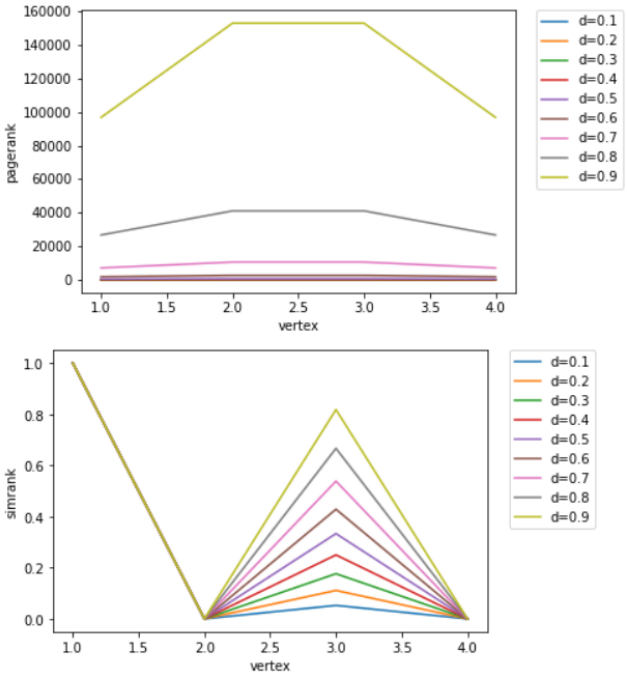
根據以下三張圖可以觀察出，在圖 G1 中隨著 damping factor 的增加，其 pagerank 隨之下降，而其 simrank 因為本來計算出來就只有(1,1)時有值，其他狀況下都是零，因此無法看出 simrank 隨著 decay factor 的變化。



在圖 G2 中隨著 damping factor 和 decay factor 的增加， pagerank 和 simrank 皆沒有變化，是因為 G2 是循環圖。



在圖 G3 中，隨著 damping factor 的增加，其 pagerank 隨之上升，其結果和 G1 相反。同時也可以觀察到，在(1,3)下，simrank 隨著 decay factor 的變化而增加。那是因為 G3 是雙向圖。



4. Effectiveness analysis

在相同迭代次數下(30 次)，從下表可以看出 pagerank 演算法平均花費的時間最短，因為是矩陣相乘所以其時間複雜度為 O(n^2)； simrank 演算法花費的時間最長，因為需要對 sim 矩陣內每一個元素進行父元素的探訪，因此其時間複雜度為 O(n^4)；因為 HIPS 演算法會同時計算出 hub 和 authority，因此雖然其時間複雜度和 pagerank 一樣為 O(n^2)，但因為 HIPS 是 O(n^2)的演算法執行兩次且其中計算相較 pagerank 較為複雜，因此 HIPS 的執行時間比 pagerank 長。

```
G1 的執行時間(分別為 HIPS、pagerank、simrank)
執行時間：0.001585 秒
執行時間：0.000404 秒
執行時間：0.003021 秒

G2 的執行時間(分別為 HIPS、pagerank、simrank)
執行時間：0.001534 秒
執行時間：0.000283 秒
執行時間：0.001881 秒

G3 的執行時間(分別為 HIPS、pagerank、simrank)
執行時間：0.001008 秒
執行時間：0.000274 秒
執行時間：0.001438 秒

G4 的執行時間(分別為 HIPS、pagerank、simrank)
執行時間：0.001967 秒
執行時間：0.000418 秒
執行時間：0.006557 秒

G5 的執行時間(分別為 HIPS、pagerank、simrank)
執行時間：0.085670 秒
執行時間：0.009102 秒
執行時間：29.540547 秒

G6 的執行時間(分別為 HIPS、pagerank)
執行時間：0.285900 秒
執行時間：0.034997 秒

Gibm 的執行時間(分別為 HIPS、pagerank)
執行時間：0.231266 秒
執行時間：0.017145 秒
```