Data Mining Project 3:
Link Analyze Practice

P76074575 資工所碩一 潘昌義

1. Introduction
2. Methods
    I. HITS
    II. PageRank
    III. SimRank
3. Implementation
    I. Result
    II. Computation performance
4. Discussion

1. Introduction

在第七章 Link Analyze 中，我們想要探討不同網站間的連結性，並透過該網站與其他網站的有向關係(例如 A 網站可以連到 B 網站這樣的特性)，觀察每個網站在網路中的廣泛程度，藉此可以判斷哪些網站是比較容易被獲取(被連結)的熱門網站，或許也可以藉此加強伺服器在某些網站上的運作。

在這個作業中，我們會依序探討以前廣為人知的 HITS 演算法(Hypertext Induced Topic Search)，以及之後由 Google 於 1988 年所提出的 PageRank 演算法(對，是以他們的創辦人之一 Larry Page 的姓命名的 XD)，還有 2002 年由 MIT 實驗室基於拓樸學提出的 SimRank 演算法。雖然前兩者現在已經落伍了，但其基礎原理還是非常值得我們去探討的。

我們這次的實驗採用兩種不同的 dataset，第一種是由教授於課堂中提供的 hw3dataset，裡面含有六張內容不同的 graph，前四張為少數節點(nodes)構成的各種有向圖，讓我們探討在不同情況下，程式的 recursive 會怎麼執行；後面兩張則讓我們討論當節點與有向關係變多時，執行的時間與迭代的次數會如何增長。

2. Methods

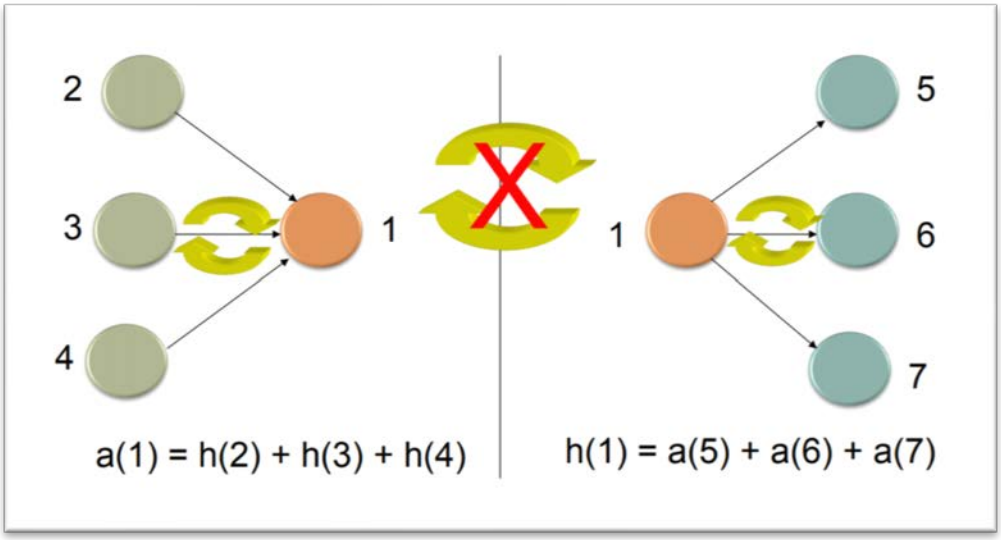
在這個環節中，我會先介紹三個演算法其演算方式，並說明為什麼這個演算法在推出的時候能夠如此成功。

I. HITS(Hypertext Induced Topic Search)

在開始之前，我還是要先聲明一下為什麼我一直寫 S 是 Search，而不是寫作 Selection，是因為我在研究演算法的時候，我拿 Selection 去做搜尋，得到的結果大概只有兩三個網站寫 Selection，剩下的網站都寫作 Search，因此我換了個比較通俗(?)的寫法…

HITS 演算法，又稱 hubs and authorities 演算法(維基百科)，是一個以每個網站被連結的強度(Authority)與對外連結的強度(Hubness)，作為評估該網站被檢索與檢索他人的程度。其計算方式非常簡單，在準備階段(或稱 Iteration 0)的時候將每個節點的 Authority 與 hubness 值都設為 1，接著在每一次的迭代(Iteration)中，將每個節點的 Authority 以“連結至該節點的每個節點的 Hubness 值之和”替代，而每個節點的 Hubness 以”被該節點連結的每個節點的 Authority 之和”替代。如果前面聽不太

懂的話沒關係，我們可以看看在課程 ppt 中提供的這張示意圖：



以比較簡易的方式，我們可以解讀為“Authority 就是上層 hubness 的和，Hubness 就是下層 Authority 的和”，因此設計程式時要不斷去確認每個節點的 Authority 與 Hubness。

最後簡單敘述下為什麼他已經被淘汰了，理由非常簡單，因為他所需計算時間太長，詳細實驗將在 Implementation 描述。

II. PageRank

現在最廣為人知的 Google，其強大的搜尋引擎是當時他崛起的一大因素，在當時他便是使用 PageRank 這個演算法，將搜尋引擎結果進行排名，讓使用者可以更容易得到他們所想找的連結網站。該演算法當時會獲得巨大的成功，除了不同於 HITS 具有更加簡潔的算法外，該算法也有著豐富的性質於其計算過程中。

過去 HITS 同時關心“從哪邊來”與“往哪邊去”兩個方向的特徵，其所需運算時間一直為電腦科學家們所詬病。在這個背景下，Google 提出了一個只關心“往哪邊去”的演算法，畢竟通常我們在逛網站的時候只會一直點進下一個網站，並不會去考慮這個網站還可以被哪些網站所連接起來，這就是 PageRank 的核心概念了。而實現方式也相當簡單，就是用高中都學過的馬可夫轉移矩陣(或稱馬可夫鏈，Markov chain)，來實現一個不斷遞移的搜尋結果，並嘗試找出最終使用者進入不同網站的機率，以判斷哪些網站是排名較前面的網站。

實現部份，我們將每個網站的起始機率 x 皆設為相等，即有 N 個網站的機率都是$\frac{1}{N}$，並製作一個進入下一個網站的轉移矩陣 B，透過不

斷運算 Bx = x'的方式，最後當 x=x'的時候，我們就知道已經算完可以回傳了。轉移矩陣的部分，參考課堂的 ppt 如下圖所示：

$$B = \begin{pmatrix} 0 & 1/5 & 1/5 & 1/5 & 1/5 & 0 & 1/5 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 1/3 & 0 & 1/3 & 0 & 0 \\ 1/4 & 0 & 1/4 & 1/4 & 0 & 1/4 & 0 \\ 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

~~(絕對不是我要吐槽，這個行列的方向我看的好不順眼阿…)~~

由於只需計算"往哪邊去"的資料，因此在運算時間上會比 HITS 演算法來得快，詳細說明會在 Implementation 的部分討論。

III. SimRank ~~(終於來到最後一個演算法了…)~~

隨著時代變遷，上述兩個傳統演算法也逐漸被人類的~~欲求不滿~~求知心給拋棄，科學家們也繼續研發更為理想快速的搜尋演算法，因此才有 SimRank 的誕生。不同於 PageRank，SimRank 探討的是"從哪邊來"，並假設"若兩個網站有相同的連結來源，則兩個網站搜尋排名的相似度應該很高"。因此我們要計算的是兩個網站 a 與 b 的 SimRank 相似度，其方程式為：

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

其中 I(x)表示連向網站 x 的所有網站的集合，C 是一個為了避免我們遭遇無窮計算所使用的阻尼係數，其範圍為(0,1)，實現過程我們留到 Implementation 部分說明。在這邊有一些計算特例需要注意，第一為"當 a=b 時，則 s(a, b)=1"，理由非常簡單，因為跟自己最像的網站就是自己；第二為"當 I(a)=0 或 I(b)=0 時，則 s(a, b)=0"，理由為"分母不得為 0"以及"沒有 in-neighbor 的話就不可能與別人有相似關係"。剩下計算的部分就是照著上面的方法，將整個 SimRank 相似度矩陣計算出來即可(因為我把他寫成一個 N*N 的矩陣了，所以我這樣解釋 XD)。

最後簡單說明一下，這個演算法能夠"活到現在"的原因是什麼，其實就是因為雖然我們都習慣"向後選新的網站"，但我們卻疏忽了一個關鍵的問題：我們怎麼來到這個網站的？透過這個問題的探討，我們就能以"從哪邊來"作為我們設計的基礎，並且得到更好的排名結果。

3. Implementation

I. Result

由於三個演算法都是透過大量迭代來計算最後的結果，因此我們最後會列出其迭代次數；又當答案為 0 的時候，有時候我們沒辦法馬上得到，甚至經過超大量的迭代後只會得到一個非常接近 0 的數字，因此我們需要設定一個 threshold 使計算到小於某個值時直接回傳 0。

此外，由於教授有在作業提及要使用 Project1 中用過的 IBM generator 生成資料測試，因此附檔中有多出兩個資料圖，一個是單向的 dirgenData.txt，另一個是雙向的 bidirgenData.txt，進行測試完的結果於表格中分別列為“IBM”與“Bi-IBM”兩個欄位。

i. HITS (實作程式：hits.py)

將每一次迭代完的結果與前一次的值相減，若小於 threshold 則視為迭代完成，回傳結果至外層。

Threshold = 0.1

	Graph1	Graph2	Graph3	Graph4	Graph5	Graph6	IBM	Bi-IBM
Count	2	2	2	2	2	2	2	2
Time	0.000	0.000	0.000	0.000	0.318	4.229	0.006	0.014

Threshold = 0.01

	Graph1	Graph2	Graph3	Graph4	Graph5	Graph6	IBM	Bi-IBM
Count	2	2	4	6	6	3	5	4
Time	0.000	0.000	0.000	0.000	0.978	6.102	0.017	0.026

不難看出 HITS 演算法在大張的 graph 上有著較差的速度，且有趣的是在雙向的圖中，雖然計算時間比較久(因為資料訊息量較多)，但 Iteration count 卻比單向的還要少。

ii. PageRank (實作程式：pagerank.py)

跟 HITS 演算法相似，但不用計算“從哪邊來”的資料，將”往哪邊去”的資料建構成一個 N*N 的矩陣做為轉移矩陣即可。要注意的是由於使用者可能放棄從該網頁提供的訊息進入下一個網站，而是隨機進入新網站，我們有一個阻尼係數 D = 0.15 表示使用者隨機選擇新網站的機率，實現如以下公式：

$$\mathrm{PR}(P_i) = \frac{d}{N} + (1-d)\sum_{P_j \in M(P_i)} \frac{PR(P_j)}{L(P_j)}$$

如此一來就會比較接近使用者真實瀏覽網頁的情況，以下附上實驗結果：

Threshold = 0.1, D = 0.15

	Graph1	Graph2	Graph3	Graph4	Graph5	Graph6	IBM	Bi-IBM
Count	7	1	1	1	1	1	5	1
Time	0.000	0.000	0.000	0.001	0.001	0.002	0.001	0.000

Threshold = 0.01, D = 0.15

	Graph1	Graph2	Graph3	Graph4	Graph5	Graph6	IBM	Bi-IBM
Count	13	1	1	1	12	1	7	1
Time	0.001	0.000	0.000	0.000	0.008	0.002	0.001	0.000

跟 HITS 演算法相比，只能說 PageRank 獲得了完勝，但有個奇特的點我們可以觀察：為什麼 Graph1 需要這麼久的時間完成呢？更奇怪的是，為什麼當 threshold 變小時，他所需要的時間也跟著加長呢？我們將這個問題留到 Discussion，先來看看 SimRank 實現結果如何吧。

iii. SimRank (實作程式：simrank.py)

與前述的 PageRank 不同，SimRank 是個只看“從哪邊來”的演算法，我們只須建構一個存放每個 node 的“上游 node 集合”即可計算兩兩 node 間的 SimRank 相似度。此外，由於跟上述兩者有個相似的問題：有沒有進行無窮迭代的可能性？因此我們需要決定該演算法的阻尼係數 C，使遇到無窮迭代時可以回傳確定為 0 的答案。由於他是一個分散型的遞迴式，因此實驗過程中我只有計時與統計每張圖共有幾個點，實驗結果如下：

Threshold = 0.1, C = 0.4

	Graph1	Graph2	Graph3	Graph4	Graph5
Graph size	6	5	4	7	469
Time	0.000	0.000	0.001	0.003	11.736

Threshold = 0.01, C = 0.4

	Graph1	Graph2	Graph3	Graph4	Graph5
Graph size	6	5	4	7	469
Time	0.000	0.000	0.000	0.001	2.773

~~我慌了行不~~，對簡單的圖而言，這個演算法還真是又快又準，但對大張的圖來說...好像太久了吧...

II. Computation Performance

在做完前面三個演算法的實驗後，我們不妨以 threshold=0.1 的情況來比較一下三者於計算時間上的差異，由於 graph6 實在太大張，因此我們這邊選擇略過：

Threshold = 0.1, D = 0.15 for PageRank, C = 0.1 for SimRank

	Graph1	Graph2	Graph3	Graph4	Graph5	IBM	Bi-IBM
Graph size	6	5	4	7	469	79	79
HITS	0.000	0.000	0.000	0.000	0.318	0.006	0.014
PageRank	0.000	0.000	0.000	0.001	0.001	0.001	0.000
SimRank	0.000	0.000	0.001	0.003	12.164	0.050	1.232

不難看出計算速度大約為 PageRank > HITS > SimRank，然而現實中比較快的兩個演算法反而是被淘汰掉的。雖然 PageRank 跟 HITS 都是單一遞迴，也就是可以寫一個 while 迴圈得到解答的方式，但是在這樣簡易的計算下，我們可能很難推得較富含網頁背後的真實意義；計算較為複雜的 SimRank 反而透過多層遞迴，得到兩兩網頁的相關性，推出比較有意義的網頁排名指標。

4. Discussion

I. 我們要怎麼樣去改善 graph1、graph2 與 graph3 中 node 1 的"hub、authority 與 PageRank 值"呢？

首先，我們要先知道三個不同的值的意義：

i. Hub：該值為其下游 Authority 值的和

ii. Authority：該值為其上游 Hub 值的和

iii. PageRank value：該值會被該點所有能連接的點影響

接著我們來看看每一張圖 node1 的情況，並講解如何改善：

i. Graph1

由於 node1 只有向後連接至 node2，因此其 hub 的值由 node2 的 authority 值決定，故我們可以添加多個(x,2)的 link 使 node1 的 hub 值上升，而 authority 的值只能透過添加(x,1)的 link 使其上升；最後 PageRank 的部分除了添加(1,1)link 製造無窮進入該頁面的可能性外，也可以多製造(x,1)的 link 使大家都會連到 node1。

ii. Graph2

這張圖比較特別的地方是它會繞一圈回來，對 hub 值我們可以添加(x,2)的 link 數量，並添加(x,5)或(x,1)的 link 來讓 authority 值上升；PageRank 則一樣是添加(1,1)或(x,1)的 link。

iii. Graph3

不同於前兩者，graph3 是個雙向的直線圖，即1 ↔ 2 ↔ 3 ↔ 4的情況，因此 hub 跟 authority 的值只要增加 node1 跟 node4 的雙向關係便可直接上升，另外 PageRank 值則是添加(1,1)的 link 即可。

II. 三個演算法有沒有需要任何限制？

在撰寫三個演算法的時候，雖然看似用簡單的遞迴式就可以解決，但實際院算後發現很容易出現"結果永不相等"的情況，因此我們需要設定一個 threshold 給這些演算法，讓他們知道在什麼時候要提前判斷已經收斂至 0，開始回傳結果而非繼續下一個迭代。有趣的是當我們給定不同的 threshold 時，我們會發現在某幾張圖中(例如 graph1.txt)迭代的次數反而變高，這就表示該圖有出現無窮迭代的情況，我們便可以判斷 threshold 在此時確實發揮了作用，讓演算法沒有一直跑下去，進而回傳非常接近正確答案的結果。

III. 若是將這些演算法套用在真實網路中的網站上，我們需要注意什麼？

我覺得有幾個點需要注意：

(1) 使用者要怎麼再次連到自己？
除了 F5 不斷刷新之外，很少會有網站會讓使用者在一個網站上不停重洗相同畫面。

(2) 隨機連線的定義？
在 PageRank 算法中，我們有討論一個讓使用者隨機進入任一網頁的機率，但原則上來說，使用者應該只會往搜尋引擎去做新的搜尋，而不會公平隨機進入任何網頁才對。

(3) 返回上一頁該如何定義？
在很多時候，我們發現這個頁面不是我們要的結果，可能會選擇返回上一頁，此時我們到底要不要將這個動作歸類為"兩個網頁含有雙向 link"的圖呢？這樣一來計算出來的結果就跟原先的會有很大的不一樣了，其中 PageRank 的值更是為因此上升許多。

IV. 在 SimRank 演算法中，阻尼係數 C 的功用？

由於 SimRank 演算法是個類似 DFS 的遞迴演算法，因此在往下一層迭代求值的過程中，我們並不知道外層的值到底多小，因此我們需要在每次做新的迭代時，先算一下"直到這層為止，SimRank Value 已經變多小了"。因此需要有一個 C 來控制，使我們可以先算算是不是已經小到可以回傳 0 了(亦即小於 threshold)，來減少演算法不斷迭代極小數值的情況。

V. 有什麼關於鏈結分析(Link Analyze)的新點子嗎？

~~我可以說沒有嗎:D~~

"從哪裡來"與"往哪邊去"的資料已經被上面三個演算以一併考慮或單獨選一個討論及實作過了，我想我們除了選擇增加觀察範圍(例如一次看兩層的關係，往上兩層與往下兩層)外，可能只能加入更多人為因素做參考了，例如調大進入搜尋引擎網站的機率(Google 等)、將書籤列(或我的最愛)放進考量、歸類網域(google.com 的網站直接歸為一類)等方式。

關係探討的內容細節，我覺得可以從 SimRank 的"兩兩網站的相似度"著墨，拓展到 N 個網站的相似度，但這樣感覺矩陣會變很大，可能比較難實現就是了。

VI. 其他討論與心得

這次報告最讓人感到心累的地方就是設計演算法了，雖然教授上課有說只要按照課程 ppt 的公式下去代入就行，但實際套下去會發現要自己寫的東西還很多，例如完全沒提到的 threshold，或者是參數的調整(SimRank 的 C)等問題。

最讓人懊惱的是資料並沒有說“每個數字都有出現過”，當我在設計演算法的矩陣時才注意到這個問題，害我把前面的 array 整個打掉重寫，改以一個 dictionary 的方式實現；寫到中間發現需要回去尋找哪個 node 在第幾行第幾列的時候，又發現一個一個搜尋會增加太多時間，使我又要再回去新增另一個 dictionary 讓演算法完全成立，根本是驚喜中的驚喜，寫到一整個很疲倦…

雖然說三個演算法的概念都很簡單，單純只是動手以紙筆操作確實容易，但若要計算大張圖的輸出(graph5 與 graph6)時，就真的需要依靠程式幫我們計算了，哪來那麼多的空間讓我們算啊。這個作業也讓我們實體見識到電腦在運算的強大(？)，更讓我們知道網頁對網頁之間的關係竟然是那麼的複雜。為了使搜尋引擎更為強大，我們要不斷的提升 Recall 率，最好的方式就是找出越容易被使用者瀏覽或需要的網頁，將其放置於搜尋結果的頂端，讓使用者能更快點擊至想要的網頁，減少搜尋所需的時間。

最後也讓我抱怨一下，網路上寫的 code 實在是叫人難以理解，寫了一大堆的函數實現一樣的事情，即便他加了一堆註解我也看不懂啊嗚嗚…