Data Mining Project 3 Report

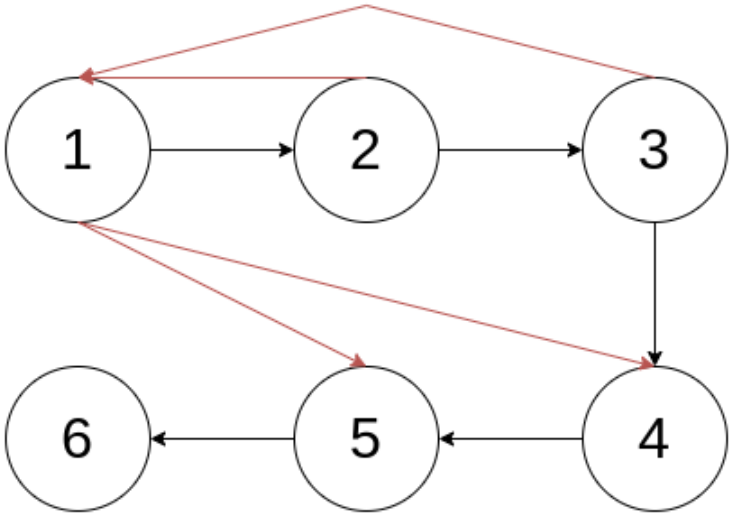
tags: 資料探勘

Author Infomation

- Name: 洪裕翔
- Grade: 資訊所碩士一年級
- Student ID: P76124215

1. Find a Way

1.1 Revised Graph 1



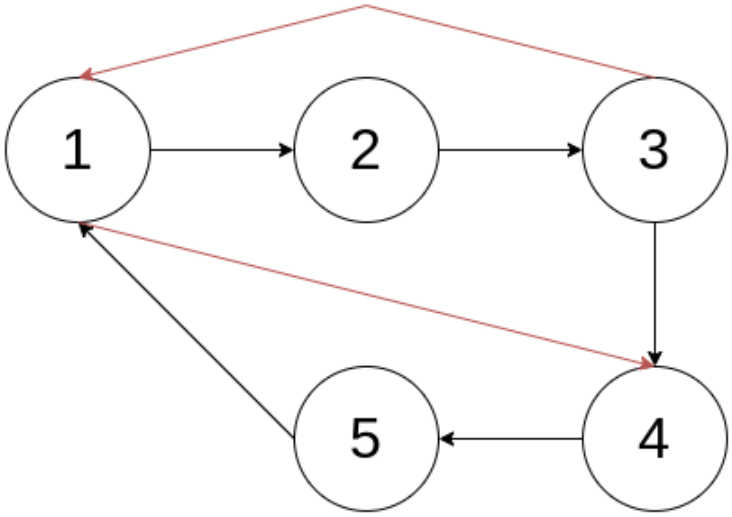
Before revised

Node	Authority	Hub	PageRank
1	0.000	0.200	0.056
2	0.200	0.200	0.107
3	0.200	0.200	0.152
4	0.200	0.200	0.193
5	0.200	0.200	0.230
6	0.200	0.000	0.263

After revised

Node	Authority	Hub	PageRank
1	0.188	0.429	0.216
2	0.187	0.143	0.101
3	0.063	0.286	0.082
4	0.312	0.143	0.138
5	0.250	0.000	0.225
6	0.000	0.000	0.239

1.2 Revised Graph 2



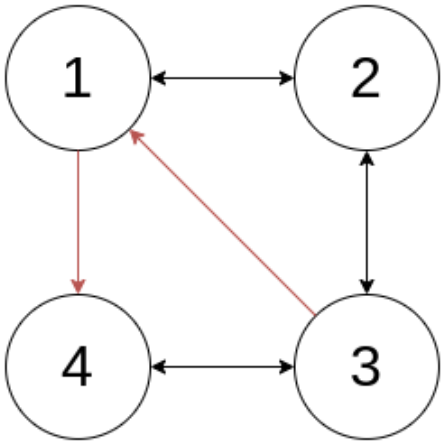
Before revised

Node	Authority	Hub	PageRank
1	0.200	0.200	0.200
2	0.200	0.200	0.200
3	0.200	0.200	0.200
4	0.200	0.200	0.200
5	0.200	0.200	0.200

After revised

Node	Authority	Hub	PageRank
1	0.357	0.357	0.279
2	0.198	0.000	0.145
3	0.000	0.445	0.151
4	0.445	0.000	0.213
5	0.000	0.198	0.212

1.3 Revised Graph 3



Before revised

Node	Authority	Hub	PageRank
1	0.191	0.191	0.172
2	0.309	0.309	0.328
3	0.309	0.309	0.328
4	0.191	0.191	0.172

After revised

Node	Authority	Hub	PageRank
1	0.262	0.322	0.224
2	0.322	0.178	0.224
3	0.093	0.453	0.328
4	0.322	0.047	0.224

2. Algorithm Description

2.1 PageRank

僅節錄與此演算法有關的核心程式碼。

PageRank 演算法簡而言之便是：通過多次迭代，計算所有點輸入點的 pagerank 總和，並正規化。
其精神在於：越重要的網站，可能會被更多網站連接，因此其僅考慮所有輸入點的資訊。

由 `pagerank` 函式可以看到，程式會做 `iteration` 次的迭代，每一次迭代都會計算所有點輸入點的 pagerank 總和，作為當前點的 pagerank，再對其正規化，使 pagerank 經過多次迭代計算後，可以收斂。

值得注意的是，PageRank 演算法引入了 `damping factor` 用於計算 pagerank，此變數的意義是計算網頁權重的因素，可以視為「用戶選擇點擊連結，繼續瀏覽網站」的機率，通過此變數，可以提昇評估網頁權重的現實有效性。

```
class Node():

    def __init__(self, name):
        self.name = name
        self.parents = []
        self.children = []
        self.pagerank = 1.0

    def update_pagerank(self, damping_factor: float, nodes_number: int):
        self.pagerank = damping_factor / nodes_number + \
            (1 - damping_factor) * sum(node.pagerank / len(node.children)
                                       for node in self.parents)


class Graph():

    def __init__(self):
        """ Initialize the graph.
        """

        self.nodes = {}

    def normalize_pagerank(self):
        """ Normalize the pagerank of each node.
        """

        total_pagerank = sum(node.pagerank for node in self.nodes.values())

        for node in self.nodes.values():
            node.pagerank /= total_pagerank


def pagerank(graph: Graph, damping_factor: float, iteration: int):
    """ Calculate the pagerank of the graph.

    Args:
        graph (Graph): The graph of the data.
        damping_factor (float): The damping factor.
        iteration (int): The number of iterations.
    """

    for _ in range(iteration):
        for node in graph.nodes.values():
            node.update_pagerank(damping_factor=damping_factor,
                                 nodes_number=len(graph.nodes))

        graph.normalize_pagerank()
```

2.2 HITS

僅節錄與此演算法有關的核心程式碼。

HITS 演算法和 PageRank 演算法有些類似，都是通過多次迭代，計算數值以排序。差別在於：HITS 演算法通過多次迭代，計算所有點輸入點的 authority 和輸出點的 hub 總和，並正規化。

由 `hits` 函式可以看到，程式會做 `iteration` 次的迭代，每一次迭代都會先計算所有點輸入點的 authority 總和，作為當前點的 authority；接著再計算所有點輸出點的 hub 總和，作為當前點的 hub；都計算完畢後，便對其正規化，使 authority 和 hub 經過多次迭代計算後，可以收斂。

```
class Node():

    def __init__(self, name):
        self.name = name
        self.parents = []
        self.children = []
        self.authority = 1.0
        self.hub = 1.0

    def update_authority(self):
        self.authority = sum(node.hub for node in self.parents)

    def update_hub(self):
        self.hub = sum(node.authority for node in self.children)


class Graph():

    def __init__(self):
        """ Initialize the graph.
        """

        self.nodes = {}

    def normalize_authority_and_hub(self):
        """ Normalize the authority and hub of each node.
        """

        total_authority = sum(node.authority for node in self.nodes.values())
        total_hub = sum(node.hub for node in self.nodes.values())

        for node in self.nodes.values():
            node.authority /= total_authority
            node.hub /= total_hub


def hits(graph: Graph, iteration: int):
    """ Calculate the authority and hub of the graph.

    Args:
        graph (Graph): The graph of the data.
        iteration (int): The number of iterations.
    """

    for _ in range(iteration):
        for node in graph.nodes.values():
            node.update_authority()

        for node in graph.nodes.values():
            node.update_hub()

        graph.normalize_authority_and_hub()
```

2.3 SimRank

僅節錄與此演算法有關的核心程式碼。

SimRank 演算法和 HITS 演算法、PageRank 演算法差異比較大，其核心概念是計算給定資料中，點之間的結構相似性，從而判斷其相似度。
基於以上敘述，SimRank 演算法在設計上會有一個紀錄點對點的 similarity 大表格，演算法在迭代的過程中，便是不斷地更新表格內的值。

由 `simrank` 函式可以看到，程式會做 `iteration` 次的迭代，每一次迭代都會先分別計算所有點和所有點的 similarity，計算的方法則在 `Similarity` 類別的 `calculate_simrank` 方法中，如果當前的兩個點相同，即一模一樣，回傳 1；若兩點中存在至少一點沒有任何輸入點，則回傳 0；如果兩點不同且兩點都有至少一個輸入點，則會基於兩點的所有輸入點，計算其結構的相似程度，並回傳最後的相似度。

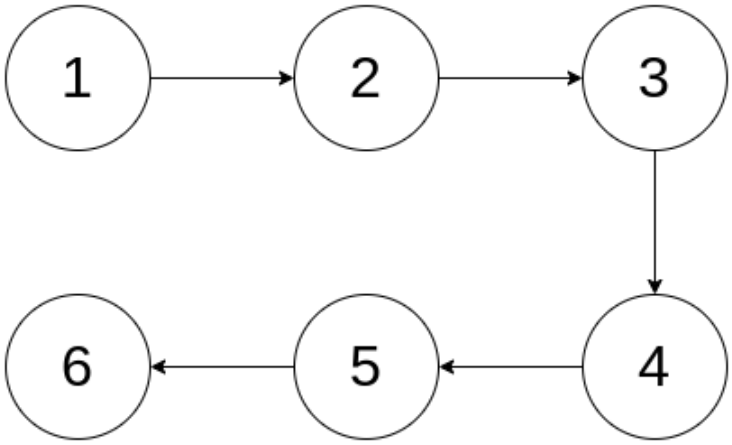
```
class Node():

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


class Graph():

    def __init__(self):
        """ Initialize the graph.
        """

        self.nodes = {}


class Similarity():

    def __init__(self, graph: Graph, decay_factor: float):
        """ Initialize the similarity class.

        Args:
            graph (Graph): The graph of the data.
            decay_factor (float): The decay factor.
        """

        self.decay_factor = decay_factor
        self.similarity = self.initialize_similarity(graph)

    def initialize_similarity(self, graph: Graph) -> List[List[float]]:
        """ Initialize the similarity matrix.

        Args:
            graph (Graph): The graph of the data.

        Returns:
            List[List[float]]: The similarity matrix.
        """

        similarity = []

        for node_1 in graph.nodes.values():
            one_row_similarity = []

            for node_2 in graph.nodes.values():
                if node_1 == node_2:
                    one_row_similarity.append(1.0)
                else:
                    one_row_similarity.append(0.0)

            similarity.append(one_row_similarity)

        return similarity

    def calculate_simrank(self, node_1: Node, node_2: Node) -> float:
        """ Calculate the simrank of the two nodes.

        Args:
```

```
            node_1 (Node): The first node.
            node_2 (Node): The second node.

        Returns:
            float: The simrank of the two nodes.
        """

        if node_1 == node_2:
            return 1.0

        if len(node_1.parents) == 0 or len(node_2.parents) == 0:
            return 0.0

        total_simrank = 0.0
        for parent_1 in node_1.parents:
            for parent_2 in node_2.parents:
                total_simrank += self.get_similarity(node_1=parent_1,
                                                     node_2=parent_2)

        return self.decay_factor * total_simrank / (len(node_1.parents) *
                                                    len(node_2.parents))

    def get_similarity(self, node_1: Node, node_2: Node) -> float:
        """ Get the similarity of the two nodes.

        Args:
            node_1 (Node): The first node.
            node_2 (Node): The second node.

        Returns:
            float: The similarity of the two nodes.
        """

        return self.similarity[int(node_1.name) - 1][int(node_2.name) - 1]

    def update_similarity(self, node_1: Node, node_2: Node, simrank: float):
        """ Update the similarity of the two nodes.

        Args:
            node_1 (Node): The first node.
            node_2 (Node): The second node.
            simrank (float): The simrank of the two nodes.
        """

        self.similarity[int(node_1.name) - 1][int(node_2.name) - 1] = round(
            number=simrank, ndigits=3)


def simrank(graph: Graph, similarity: Similarity, iteration: int):
    """ Calculate the simrank of the graph.

    Args:
        graph (Graph): The graph of the data.
        similarity (Similarity): The similarity class.
        iteration (int): The number of iterations.
    """

    for _ in range(iteration):
        for node_1 in graph.nodes.values():
```

```
            for node_2 in graph.nodes.values():
                simrank = similarity.calculate_simrank(node_1=node_1,
                                                       node_2=node_2)
                similarity.update_similarity(node_1=node_1,
                                             node_2=node_2,
                                             simrank=simrank)
```

3. Result Analysis and Discussion

3.1 Graph 1



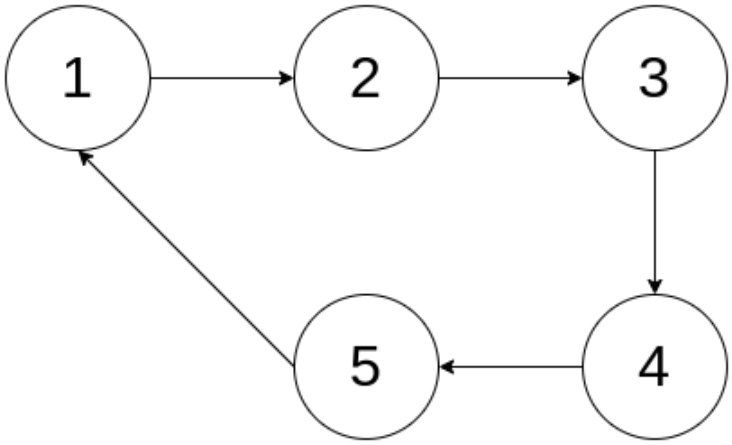
Node	Authority	Hub	PageRank
1	0.000	0.200	0.056
2	0.200	0.200	0.107
3	0.200	0.200	0.152
4	0.200	0.200	0.193
5	0.200	0.200	0.230
6	0.200	0.000	0.263

```
SimRank

1.000 0.000 0.000 0.000 0.000 0.000
0.000 1.000 0.000 0.000 0.000 0.000
0.000 0.000 1.000 0.000 0.000 0.000
0.000 0.000 0.000 1.000 0.000 0.000
0.000 0.000 0.000 0.000 1.000 0.000
0.000 0.000 0.000 0.000 0.000 1.000
```

1. 觀察 Authority 可以發現，除了 Node 1 之外，所有點的 Authority 皆為 0.2。這是由於 Node 1 沒有輸入點，而其他的點由於依序連接，所以平均了整體的 Authority。
2. Hub 則和 Authority 相反，除了 Node 6 之外，所有點的 Hub 皆為 0.2。這是由於 Node 6 沒有輸出點，而其他的點由於依序連接，所以平均了整體的 Hub。

3. 可以發現 PageRank 隨著 Node 的編號變大而增大，這是因為點之間的連接有依序，也就是 1 -> 2、2 -> 3、...，Node 1 沒有任何連結連入，而 Node 6 是整串連結的終點，所以導致這樣的結果。
4. SimRank 的結果呈現對角線值皆為 1，其餘皆為 0，這是因為不存在任何一點，其向外的連結指向兩個以上不同的點。

3.2 Graph 2

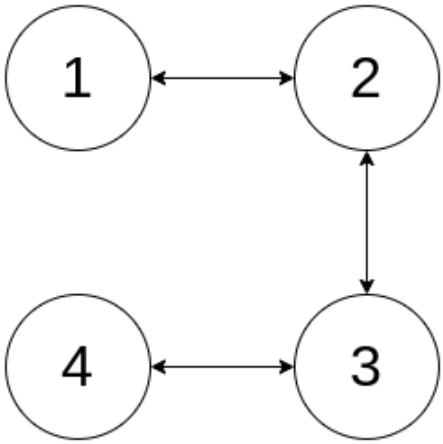


Node	Authority	Hub	PageRank
1	0.200	0.200	0.200
2	0.200	0.200	0.200
3	0.200	0.200	0.200
4	0.200	0.200	0.200
5	0.200	0.200	0.200

```
SimRank

1.000 0.000 0.000 0.000 0.000
0.000 1.000 0.000 0.000 0.000
0.000 0.000 1.000 0.000 0.000
0.000 0.000 0.000 1.000 0.000
0.000 0.000 0.000 0.000 1.000
```

1. 觀察 Authority 可以發現，所有點的 Authority 皆為 0.2。這是由於所有點依序連接，且形成了閉環，所以所有點平均了整體的 Authority。
2. Hub 和 Authority 相同，所有點的 Hub 皆為 0.2，所有點依序連接，且形成了閉環，所以所有點平均了整體的 Hub。
3. PageRank 的結果也和 Authority、Hub 相同，所有點的 PageRank 皆為 0.2。由於連結為閉環，且每個點依序連結，故所有點的權重接相同。
4. SimRank 的結果呈現對角線值皆為 1，其餘皆為 0，這是因為不存在任何一點，其向外的連結指向兩個以上不同的點。

3.3 Graph 3



Node	Authority	Hub	PageRank
1	0.191	0.191	0.172
2	0.309	0.309	0.328
3	0.309	0.309	0.328
4	0.191	0.191	0.172

```
SimRank

1.000 0.000 0.538 0.000
0.000 1.000 0.000 0.538
0.538 0.000 1.000 0.000
0.000 0.538 0.000 1.000
```

1. 觀察 Authority 可以發現，Node 1 和 Node 4 相同（0.191）、Node 2 和 Node 3 相同（0.309）。這是由於 Node 2、Node 3 比 Node 1、Node 4 多了一個輸入的連結，因此得到比較高的 Authority。
2. Hub 則和 Authority 相同，Node 1 和 Node 4 相同（0.191）、Node 2 和 Node 3 相同（0.309）。這是由於 Node 2、Node 3 比 Node 1、Node 4 多了一個輸出的連結，因此得到比較高的 Hub。
3. PageRank 和 Authority、Hub 有類似的結果，原因與 Authority 相同，是因為Node 2、Node 3 比 Node 1、Node 4 多了一個輸入的連結，因此得到比較高的 PageRank。
4. SimRank 的結果呈現對角線值皆為 1，(0, 2)、(2, 0)、(1, 3)、(3, 1) 則為 0.538，這是因為 Node 1 和 Node 3 和 Node 2 都有連結，因此存在一定的相似性，故不為 1，但也不完全相同（Node 1 沒有和其他點相連，但 Node 3 和 Node 4 有連結），故不為 0。Node 2 和 Node 4 同理。

3.4 Damping Factor

我以 Graph 3 為資料，調整 Damping Factor，觀察結果。

如我在 2.1 節所說，Damping Factor 可以被視為「用戶選擇點擊連結，繼續瀏覽網站」的機率。因此，當我們提高 Damping Factor，就代表著提高了隨機性，從理論上推斷，分數應該會越來越趨近於均勻分佈，從下面的三個實驗結果也可以發現，不同 Node 之間的 PageRank 越來越接近，有符

合前面的推測。

- Damping Factor: 0.1
  PageRank = [0.172, 0.328, 0.328, 0.172]

- Damping Factor: 0.5
  PageRank = [0.2, 0.3, 0.3, 0.2]

- Damping Factor: 0.9
  PageRank = [0.238, 0.262, 0.262, 0.238]

3.5 Decay Factor

我以 Graph 3 為資料，調整 Decay Factor，觀察結果。

Decay Factor 決定的是兩點之間相似性的衰減程度，理論上來說，其值越大，衰減的值會越小。
從下面的三個實驗結果可以發現，當 Decay Factor 越大時，相同位置的相似性會越高，是符合理論的結果。

- Decay Factor: 0.7
  SimRank = [[1.0, 0.0, 0.538, 0.0], [0.0, 1.0, 0.0, 0.538], [0.538, 0.0, 1.0, 0.0], [0.0, 0.538, 0.0, 1.0]]

- Decay Factor: 0.85
  SimRank = [[1.0, 0.0, 0.739, 0.0], [0.0, 1.0, 0.0, 0.739], [0.739, 0.0, 1.0, 0.0], [0.0, 0.739, 0.0, 1.0]]

- Decay Factor: 1
  SimRank = [[1.0, 0.0, 1.0, 0.0], [0.0, 1.0, 0.0, 1.0], [1.0, 0.0, 1.0, 0.0], [0.0, 1.0, 0.0, 1.0]]

4. Effectiveness Analysis

4.1 PageRank

graph_1	graph_2	graph_3	graph_4	graph_5	graph_6	ibm-5000
0.00025	0.00023	0.00020	0.00032	0.01219	0.02909	0.02069

4.2 HITS

graph_1	graph_2	graph_3	graph_4	graph_5	graph_6	ibm-5000
0.00028	0.00024	0.00021	0.00035	0.01053	0.03361	0.02281

4.3 SimRank

graph_1	graph_2	graph_3	graph_4	graph_5	graph_6	ibm-5000
0.00195	0.00146	0.00114	0.00552	14.90283		

我的執行環境為 Ubuntu 22.04 LTS，處理器為 AMD Ryzen 7 5800x。
結果的時間單位皆為秒，精確位數取到小數點後第五位。

首先比較 PageRank 演算法和 HITS 演算法的結果，能發現在大部分的資料（除了 graph_5）下，PageRank 演算法都比 HITS 演算法稍快一些，我認為這是因為 HITS 演算法在計算 authority 和 hub 時，需要不斷地計算輸入點（parent node）的 hub 總和與輸出點（child node）的 authority 總和；而 PageRank 演算法則只需要取輸入點的 pagerank，因此可以節省需要的執行時間。

另外，可以很清楚地觀察到，SimRank 演算法的時間在所有資料上，都遠大於 PageRank 演算法和 HITS 演算法。這是由於 SimRank 演算法在計算 similarity 時，需要對連結中兩個點的輸入點（parent node）都分別取值、計算總和，這會使 SimRank 演算法面對連結數巨大的資料時，需要的執行時間非常久。