Game Theory Homework 1

311551094 廖昱瑋 資科工所碩一

1. Simulation Environment (WS model)

首先，先介紹如何建置 WS model，這個模擬環境會用在接下來 requirement 1-1、1-2、2 的部分。以 python 中 networkx package 提供的 "watts_strogatz_graph" function 建立 WS model，其中參數 n 代表 node 數目，k 代表初始時每個 node 的 neighbors 數目，p 代表 rewiring 的機率。

接下來的題目皆設置：$n = 30$、$k = 4$、$p = 0 \quad 0.2 \quad 0.4 \quad 0.6 \quad 0.8$。並且每種 rewiring 機率要各跑 100 次，以呈現之後實作部分的 performance。

```
NODE_NUM = 30
LINK_NUM = 4
PROBABILITY = [p*0.2 for p in range(5)]
```

圖 1. WS model 的參數

```
for p in range(5):
    for i in range(100):
        #create watts_strogatz model with 30 nodes, 4 links for each node initially
        G = nx.watts_strogatz_graph(n = NODE_NUM, k = LINK_NUM, p = PROBABILITY[p])
```

圖 2. 以 networkx 建置 WS model

2. Requirement 1-1 (Weighted MIS Game)

2.1 Utility Function and Best Response

Weighted MIS Game 的每個 node 都有一個整數 weight，介於[0, 29]。我定義其 priority function：

$$\frac{W(p_i)}{W(p_i) + \sum_{p_j \in N_i} W(p_j)}$$

其中，p_i 為 node i、N_i 為 p_i 的 open neighbor、$W(p_i)$為 node i 的 weight。

Utility function：

$$u_i(C) = \sum_{p_j \in L_i} \omega(c_i, c_j) + c_i$$

其中，$\omega(c_i, c_j) = -\alpha c_i c_j$、$\alpha$為大於 1 的常數、$L_i$ 為 priority 大於等於 p_i priority 的 neighbors。

由以上 utility function，我們可以推論出 best response：

$$BR_i(c_{-i}) = \begin{cases} 0, & if \exists p_j \in L_i, c_j = 1 \\ 1, & otherwise \end{cases}$$

2.2 Code

在 "add_node_weight" function randomly assign[0, 29]範圍的 weight 給每個 node，然後藉由 "calculate_node_priority" function 以上述公式計算各 node 的 priority。並以 "initialize_strategy_profile" function randomly 給予每個 node strategy∈{0, 1}。

```python
#add random node weight to the WS model
def add_node_weight(G, node_num):
    weights = {}
    for node in range(node_num):
        weights[node] = random.randint(0,node_num-1)
    nx.set_node_attributes(G, weights, name='weight')

    return
```

圖 3. “add_node_weight” function

```python
#calculate prioity for each node
def calculate_node_priority(G, node_num):
    priorities = {}
    for node in range(node_num):
        neighors_weight = G.nodes[node]['weight']
        for neighbor in G.neighbors(node):
            neighors_weight += G.nodes[neighbor]['weight']
        if neighors_weight != 0:
            priorities[node] = G.nodes[node]['weight'] / neighors_weight
        else:
            priorities[node] = 0
    nx.set_node_attributes(G, priorities, name='priority')

    return
```

圖 4. “calculate_node_priority” function

```python
#randomly initialize strategy profile
def initialize_strategy_profile(G, node_num):
    strategy_profile = {}
    for node in range(node_num):
        strategy_profile[node] = random.randint(0,1)
    nx.set_node_attributes(G, strategy_profile, name='strategy')

    return
```

圖 5. “initialize_strategy_profile” function

接著，建立一個迴圈，每次迴圈隨機挑選一個可以增加自身 utility 的 node，並把它的 strategy 改為其 best response，一直重複執行，直至沒有任何 node 可以透過更改 strategy 增加其 utility。

```python
#randomly select a node that its best response is not its strategy, iterate until graph is MIS
move_count = 0
while True:
    waiting_nodes = find_waiting_node_MIS(G, NODE_NUM)
    if not waiting_nodes:
        break
    else:
        player = random.choice(list(waiting_nodes))
        G.nodes[player]['strategy'] = waiting_nodes[player]
        move_count += 1

average_move_count[p] += move_count
average_total_weight[p] += calculate_total_weight(G, NODE_NUM)
```

圖 6. Weighted MIS Game 主迴圈架構

透過 “find_waiting_node_MIS” function 尋找可以更改 strategy 增加其 utility 的 node set。“find_waiting_node_MIS” function 中運用上述 best response 公式，先得到每個 node 的 best response，並與它當前 strategy 做比較，若不相同，就把 node number 當成 key；best response 當成 value 存進 “waiting_nodes” dictionary 中。

```python
#find node set that its strategy is not best response in weighted MIS, store node as key, best response as value
def find_waiting_node_MIS(G, node_num):
    waiting_nodes = {}
    for node in range(node_num):
        priority = G.nodes[node]['priority']
        strategy = G.nodes[node]['strategy']
        best_response = 1
        for neighbor in G.neighbors(node):
            if G.nodes[neighbor]['priority'] > priority and G.nodes[neighbor]['strategy'] == 1:
                best_response = 0
                break
        if strategy != best_response:
            waiting_nodes[node] = best_response

    return waiting_nodes
```

圖 7. “find_waiting_node_MIS” function

在不同 rewiring probability 情形下，以所有 picked node 的總 weight 及平均每個 node 進行 strategy 改變的次數分析 performance。每個 rewiring probability 做 100 次再取平均。當中，使用 “calculate_total_weight” function 計算圖中 picked node 的總 weight。

```python
#calculate total weight of selected nodes in MIS
def calculate_total_weight(G, node_num):
    total_weight = 0
    for node in range(node_num):
        if G.nodes[node]['strategy'] == 1:
            total_weight += G.nodes[node]['weight']

    return total_weight
```

圖 8. “calculate_total_weight” function

最後，先將 node 移動次數及總 weight 取平均，並把 performance 以折線圖呈現，並用 “print_graph_MIS” function 畫一張 Weighted MIS graph 觀看結果的範例。

```
average_move_count /= (100 * NODE_NUM)
average_total_weight /= 100
```

圖 9. 將 node 移動次數及總 weight 取平均

```
plt.figure(figsize = (10, 5))
plt.subplot(1, 2, 1)
plt.plot(PROBABILITY,average_total_weight)
plt.title('Average total weight')
plt.xlabel('Link Rewiring Probability')
plt.ylabel('Average total weight')

plt.subplot(1, 2, 2)
plt.plot(PROBABILITY,average_move_count)
plt.title('Average number of moves per node')
plt.xlabel('Link Rewiring Probability')
plt.ylabel('Average number of moves per node')
plt.show()
```

圖 10. 畫 Weighted MIS performance 折線圖

```
#print weighted MIS graph
def print_graph_MIS(G):
    label = {n: str(n) + ': ' + str(G.nodes[n]['weight']) for n in G.nodes}
    color = [G.nodes[n]['strategy'] for n in G.nodes]
    cmap = colors.ListedColormap(['g','r'])
    pos = nx.circular_layout(G)
    plt.figure(figsize = (7.5, 7.5))
    nx.draw_networkx(G, pos, labels = label, node_color = color, cmap = cmap)
    plt.show()

    return
```

圖 11. “print_graph_MIS” function

2.3 Result

由圖 12.可以得知當 rewiring probability 越大時，總 weight 會越大，因為 graph 被 mixed 得更好，有助於找到更多 node 符合 independence 性質。當 rewiring probability 越大時，平均每個 node 進行 strategy 改變的次數大致有變多趨勢，但其實 y 軸數值差異太小，所以趨勢方向不是很明確。

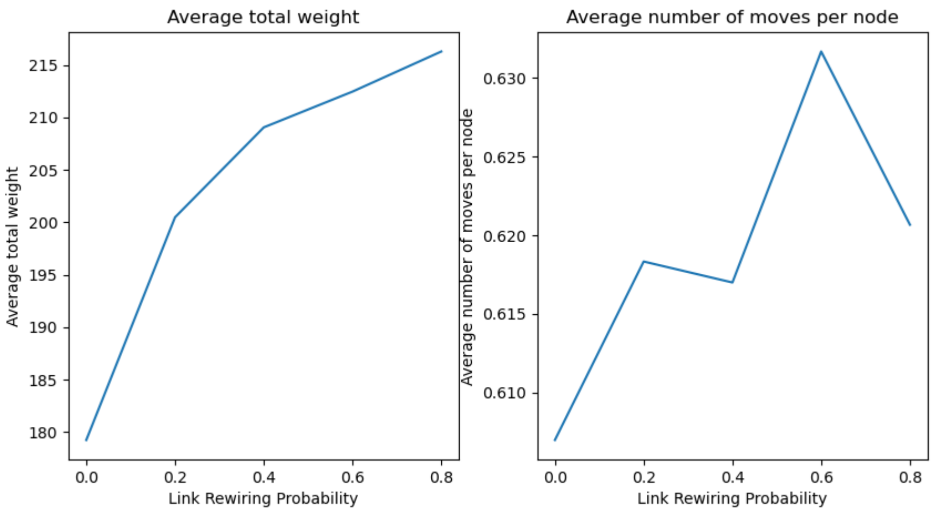

圖 12. Weighted MIS performance

圖 13.中每個 node 上的 label x:y，x 代表 node 編號；y 代表 node weight。紅點為 strategy = 1 的點；綠點為 strategy = 0 的點。

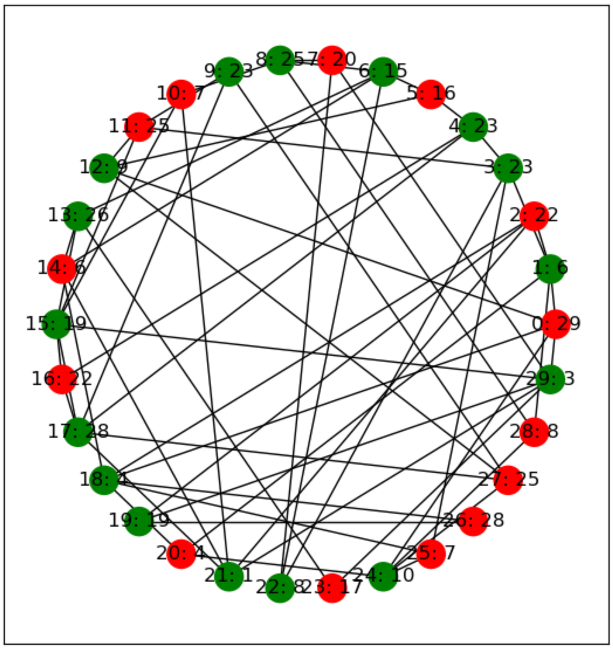

圖 13. Weighted MIS graph example

3. Requirement 1-2 (Symmetric MDS-based IDS Game)

3.1 Utility Function and Best Response

Utility function：

Let $M_i = N_i \cup \{p_i\}$. Define $v_i(C) = \sum_{p_j \in M_i} c_j$

Let $\alpha > 1$ be a constant. Define $g_i(C)$ as

$$g_i(C) = \begin{cases} \alpha & \text{if } v_i(C) = 1 \\ 0 & \text{otherwise,} \end{cases}$$

gain of dominance

Let $\gamma > n\alpha$ be a constant. Define

$$w_i(C) = \sum_{p_j \in N_i} c_i c_j \gamma,$$

penalty of violating independence

Let $0 < \beta < \alpha$. p_i's utility:

$$u_i(C) = \begin{cases} \boxed{\left(\sum_{p_j \in M_i} g_j(C)\right)} - \beta - \boxed{w_i(C)} & \text{if } c_i = 1 \\ 0 & \text{otherwise,} \end{cases}$$

由以上 utility function，我們可以推論出 best response：

$$BR_i(c_{-i}) = \begin{cases} 0, & if\, \exists p_j \in N_i, c_j = 1\ or\ \forall p_j \in M_i, v_j(c_{-i}) \geq 1 \\ 1, & otherwise \end{cases}$$

3.2 Code

先以 requirement1-1 用過的“initialize_strategy_profile” function randomly 給予每個 node strategy∈{0, 1}。

主迴圈架構與 requirement1-1 相同，只差在尋找可以更改 strategy 增加其 utility 的 node set 是用 Symmetric MDS-based IDS Game 的 best response 公式，定義於 “find_waiting_node_MDS_based_IDS” function。

並且因這裡的 node 沒有 weight，performance 改以 cardinality 測量，計算方式寫在 “calculate_cardinality” function。

```
#randomly select a node that its best response is not its strategy, iterate until graph is MIS
move_count = 0
while True:
    waiting_nodes = find_waiting_node_MDS_based_IDS(G, NODE_NUM)
    if not waiting_nodes:
        break
    else:
        player = random.choice(list(waiting_nodes))
        G.nodes[player]['strategy'] = waiting_nodes[player]
        move_count += 1

average_move_count[p] += move_count
average_cardinality[p] += calculate_cardinality(G, NODE_NUM)
```

圖 14. Symmetric MDS-based IDS Game 主迴圈架構

```python
#find node set that its strategy is not best response in symmetric MDS based IDS, store node as key, best response as value
def find_waiting_node_MDS_based_IDS(G, node_num):
    waiting_nodes = {}
    for node in range(node_num):
        strategy = G.nodes[node]['strategy']
        best_response = 1

        #check dominate
        dominate_condition = 1
        vj = 0
        for neighbor_node in G.neighbors(node):
            vj += G.nodes[neighbor_node]['strategy']

        if vj == 0:
            dominate_condition = 0
        else:
            for neighbor_node in G.neighbors(node):
                vj = 0
                for neighbor_p in G.neighbors(neighbor_node):
                    vj += G.nodes[neighbor_p]['strategy']
                if vj == 0:
                    dominate_condition = 0
                    break
```

圖 15. “find_waiting_node_MDS_based_IDS” function – (1)

```python
        #check independence
        independence_condition = 0
        for neighbor_node in G.neighbors(node):
            if G.nodes[neighbor_node]['strategy'] == 1:
                independence_condition = 1
                break

        #best response
        if dominate_condition or independence_condition:
            best_response = 0

        if strategy != best_response:
            waiting_nodes[node] = best_response

    return waiting_nodes
```

圖 16. “find_waiting_node_MDS_based_IDS” function – (2)

```python
def calculate_cardinality(G, node_num):
    total_node = 0
    for node in range(node_num):
        if G.nodes[node]['strategy'] == 1:
            total_node += 1

    return total_node
```

圖 17. “calculate_cardinality” function

最後，先將 node 移動次數及 cardinality 取平均，並把 performance 以折線圖呈現，並用 “print_graph_MDS_based_IDS” function 畫一張 Symmetric MDS-based IDS Game graph 觀看結果的範例。

```
average_move_count /= (100 * NODE_NUM)
average_cardinality /= 100
```

圖 18. 將 node 移動次數及 cardinality 取平均

```
plt.figure(figsize = (10, 5))
plt.subplot(1, 2, 1)
plt.plot(PROBABILITY,average_cardinality)
plt.title('Average Size of Symmetric MDS-based IDS')
plt.xlabel('Link Rewiring Probability')
plt.ylabel('Cardinality of Symmetric MDS-based IDS')

plt.subplot(1, 2, 2)
plt.plot(PROBABILITY,average_move_count)
plt.title('Average number of moves per node')
plt.xlabel('Link Rewiring Probability')
plt.ylabel('Average number of moves per node')
plt.show()
```

圖 19. 畫 Symmetric MDS-based IDS Game performance 折線圖

```
#print symmetric MDS-based IDS graph
def print_graph_MDS_based_IDS(G):
    color = [G.nodes[n]['strategy'] for n in G.nodes]
    cmap = colors.ListedColormap(['g','r'])
    pos = nx.circular_layout(G)
    plt.figure(figsize = (7.5, 7.5))
    nx.draw_networkx(G, pos, node_color = color, cmap = cmap)
    plt.show()

    return
```

圖 20. “print_graph_MDS_based_IDS” function

3.3 Result

由圖 21.可以得知當 rewiring probability 越大時，cardinality 會越大，因為 graph 被 mixed 得更好，有助於找到更多 node 符合 independence 及 dominance 性質。當 rewiring probability 越大時，平均每個 node 進行 strategy 改變的次數越大，因為圖變得更沒規則性，要花更多步驟找到 Nash Equilibrium。

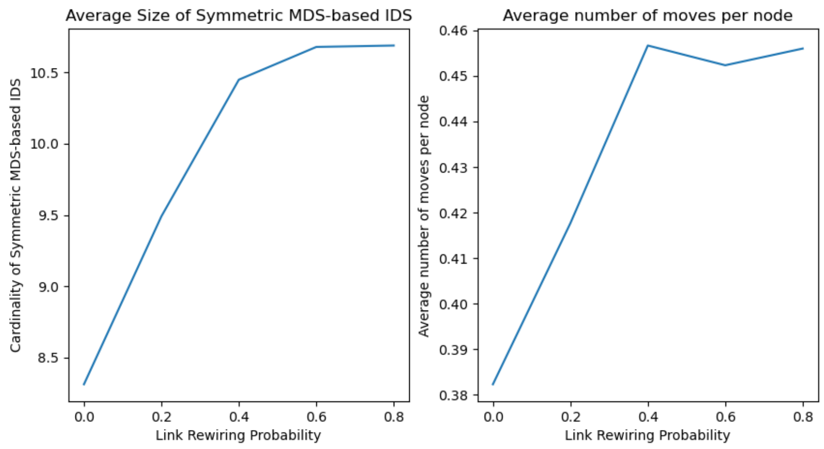

圖 21. Symmetric MDS-based IDS Game performance

圖 22.中每個 node 上數字代表 node 編號。紅點為 strategy=1 的點；綠點為 strategy=0 的點。

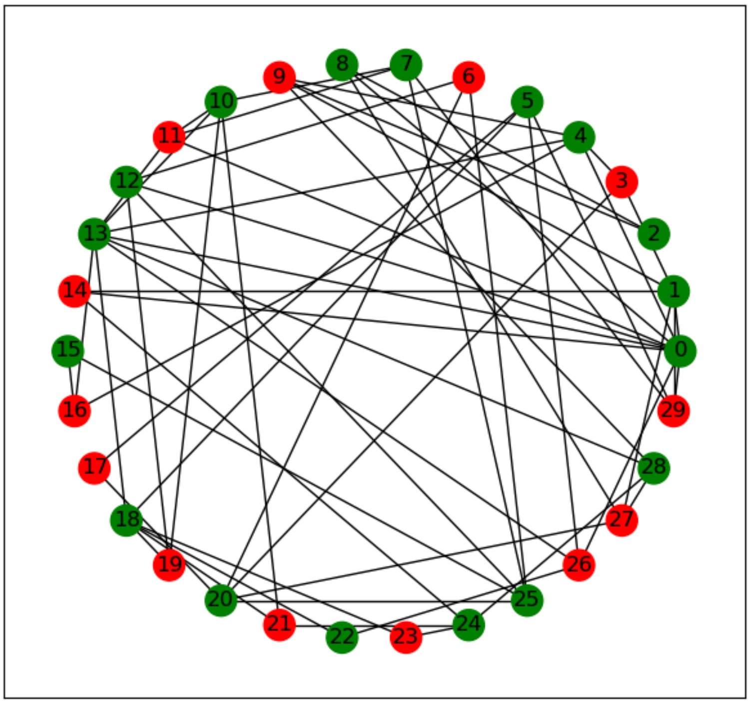

圖 22. Symmetric MDS-based IDS Game example

4. Requirement 2 (Matching Game)

4.1 Utility Function and Best Response

Utility function：

$$u_i(C) = \begin{cases} 2, & if\, c_i = j \wedge c_j = i \\ 1, & if\, c_i = j \wedge c_j = null \\ 0, & if\, c_i = null \\ -1, & if\, c_i = j \wedge c_j = k \end{cases}$$

其中$p_j \in N_i$、$k = p_k \in N_j, k \neq i \neq null$。

當 node i 是 matched 時，給予最高 utility 2；當 node i 選擇的對象並沒有選任何人時，給予一點獎賞 utility 為 1：當 node i 沒選擇任何其他點時，給予 utility 0；當 node i 選擇的對象選了其他的 node 而非 i 時，給它懲罰，所以 utility 設-1。

由以上 utility function，我們可以推論出 best response：

$$BR_i(c_{-i}) = \begin{cases} j, & if \exists p_j \in N_i, c_j = i \\ \omega(p_i), & if \forall p_j \in N_i, c_j = null \\ null, & otherwise \end{cases}$$

其中$\omega(p_i)$為隨機挑選$p_j \in N_i$中的任一 node。

當有 neighbor node 選擇 node i 時，best response 是選擇回去，形成 matched pair；當沒有 neighbor node 選擇 node i，但 neighbor node 全部都沒選擇任何 node 時，best response 就隨機挑一個 neighbor 選，因為它的 neighbor 還有機會跟它形成 matched pair；在其餘的狀況，best response 則為不選擇任何點。

4.2 Code

為了撰寫程式方便，我定義 null strategy 為-1。先以 “initialize_strategy_profile” function 隨機給予每個 node strategy$\in \{N_i, -1\}$。

這邊的“initialize_strategy_profile” function 跟 requirement 1 的不一樣。

```
#randomly initialize strategy profile
def initialize_strategy_profile(G, node_num):
    strategy_profile = {}
    for node in range(node_num):
        neighbor_nodes = [-1]   #define null stratrgy as -1
        for neighbor in G.neighbors(node):
            neighbor_nodes.append(neighbor)
        strategy_profile[node] = random.choice(neighbor_nodes)
    nx.set_node_attributes(G, strategy_profile, name='strategy')

    return
```

圖 23. “initialize_strategy_profile” function

主迴圈架構與 requirement 1-2 相同，只差在尋找可以更改 strategy 增加其 utility 的 node set 是用 Matching Game 的 best response 公式，定義於 “find_waiting_nodes” function。

performance 改以 number of matched pairs 測量，計算方式寫在 “calculate_matched_pair” function。並且在“calculate_matched_pair” function 中，順便標記 matched pairs 的 nodes 跟 weights，以利之後畫 example 圖。

```python
        #randomly select a node that its best response is not its strategy, iterate until graph is MIS
        move_count = 0
        while True:
            waiting_nodes = find_waiting_nodes(G, NODE_NUM)
            if not waiting_nodes:
                break
            else:
                player = random.choice(list(waiting_nodes))
                G.nodes[player]['strategy'] = waiting_nodes[player]
                move_count += 1

        average_move_count[p] += move_count
        average_matched_pair[p] += calculate_matched_pair(G)

average_move_count /= (100 * NODE_NUM)
average_matched_pair /= 100
```

圖 24. Matching Game 主迴圈架構

```python
#find node set that its strategy is not best response in matching game, store node as key, best response as value
def find_waiting_nodes(G, node_num):
    waiting_nodes = {}
    for node in range(node_num):
        cur_strategy = G.nodes[node]['strategy']

        if cur_strategy == -1 or G.nodes[cur_strategy]['strategy'] != node:   #If it is cuurently not matched
            has_null_neighbor = False
            best_response = -1
            null_neighbor = -1

            for neighbor in G.neighbors(node):
                if G.nodes[neighbor]['strategy'] == node:   #If it can have matched pairs
                    best_response = neighbor
                    break
                elif G.nodes[neighbor]['strategy'] == -1:   #Store the information whether it has any neighbor that its strategy is nul
                    null_neighbor = neighbor
                    has_null_neighbor = True
```

圖 25. “find_waiting_nodes” function – (1)

```python
            if best_response == -1 and has_null_neighbor:   #It can not find matched pair but has neighbor that its strategy is null
                best_response = null_neighbor

            if cur_strategy != best_response:
                waiting_nodes[node] = best_response

    return waiting_nodes
```

圖 26. “find_waiting_nodes” function – (2)

```python
#calculate total number of matched pairs and set pairs' color and edge weight
def calculate_matched_pair(G):
    matched_pair = 0
    for node in G.nodes:
        G.nodes[node]['matched'] = 0

    for edge in G.edges:
        node_1 = edge[0]
        node_2 = edge[1]
        G[node_1][node_2]['weight'] = 1
        G[node_1][node_2]['color'] = 'k'
        if G.nodes[node_1]['strategy'] == node_2 and G.nodes[node_2]['strategy'] == node_1:
            matched_pair += 1
            G[node_1][node_2]['weight'] = 2
            G[node_1][node_2]['color'] = 'r'
            G.nodes[node_1]['matched'] = 1
            G.nodes[node_2]['matched'] = 1

    return matched_pair
```

圖 27. “calculate_matched_pair” function

最後，先將 node 移動次數及 matched pairs 數目取平均，並把 performance 以折線圖呈現，並用 “print_graph” function 畫一張 Matching Game graph 觀看結果的範例。

```
average_move_count /= (100 * NODE_NUM)
average_matched_pair /= 100
```

圖 28. 將 node 移動次數及 matched pairs 數目取平均

```
plt.figure(figsize = (10, 5))
plt.subplot(1, 2, 1)
plt.plot(PROBABILITY,average_matched_pair)
plt.title('Average Number of Matched Pairs')
plt.xlabel('Link Rewiring Probability')
plt.ylabel('Average Number of Matched Pairs')

plt.subplot(1, 2, 2)
plt.plot(PROBABILITY,average_move_count)
plt.title('Average Number of Moves per Node')
plt.xlabel('Link Rewiring Probability')
plt.ylabel('Average Number of Moves per Node')
plt.show()
```

圖 29. 畫 Matching Game performance 折線圖

```
#print graph
def print_graph(G):
    weights = [G[u][v]['weight'] for u,v in G.edges]
    edge_colors = [G[u][v]['color'] for u,v in G.edges]
    node_colors = [G.nodes[n]['matched'] for n in G.nodes]
    cmap = colors.ListedColormap(['y','r'])
    pos = nx.circular_layout(G)
    plt.figure(figsize = (7.5, 7.5))
    nx.draw_networkx(G, pos, width = weights, edge_color = edge_colors, node_color = node_colors, cmap = cmap)
    plt.show()

    return
```

圖 30. “print_graph” function

4.3 Result

由圖 31.可以得知當 rewiring probability 越大時，number of matched pairs 會越小，因為 graph 被 mixed 得更好，edges 雜亂的連來連過去，會降低獨立 pairs 的數目。當 rewiring probability 越大時，平均每個 node 進行 strategy 改變的次數越小，因為圖變得更沒規則性，沒有那麼多 matched pairs，所以可以些微降低 strategy 改變的次數。

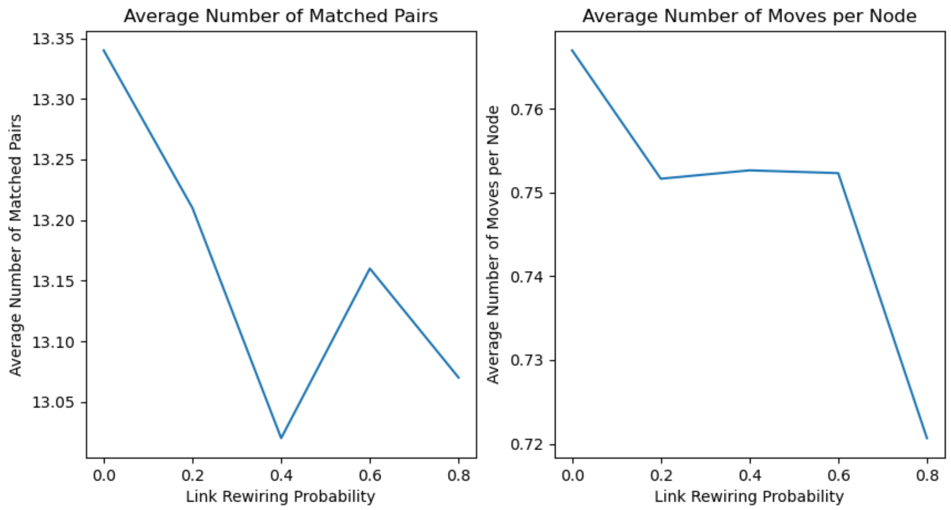

圖 31. Matching Game performance

圖 32.中每個 node 上數字代表 node 編號。紅點為有成功 matched 的點；黃點為 unmatched 的點。紅色的 edge 為 matched pairs 的 edge；黑色的 edge 為 unmatched pairs 的 edge

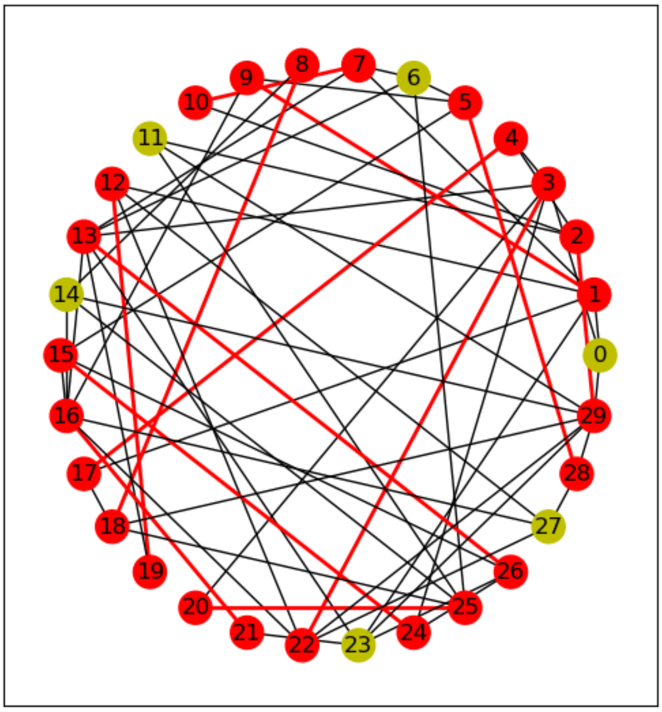

圖 32. Matching Game example