Requirement 1-1: Weighted MIS Game

Code Explanation

```
# initialize the graph based on input
def graph_init(node_num, relations):
    G = nx.Graph()
    for i in range(0, node_num):
        G.add_node(i)
    for i, relation in enumerate(relations):
        for j, r in enumerate(relation):
            if r == '1':
                G.add_edge(i, j)
    return G
```

To start simulating the game, we have to create the graph based on input first. The function will create the graph using the given parameters, so it can handle any random WS model.

```
# randomly initialize the node weights range in [0, N-1]
def node_weights_init(G):
    #weights = {i: w for i, w in enumerate(random.sample(range(node_num), node_num))} # random and unquie (old one)
    weights = {i: i+1 for i in range(node_num)} # node i has weight i
    nx.set_node_attributes(G, weights, name = 'weights')
```

To initialize game state, the function node_weights_init() is used to assign weight for all nodes. The weight value is the same as its index.

```
# priority function for weighted MIS game
def node_priority_init(G):
    priority = {}
    for i in range(node_num):
        priority[i] = G.nodes[i]['weights'] / (len(list(G.neighbors(i))) + 1)
    nx.set_node_attributes(G, priority, name = 'priority')
```

After the weight is selected, the function node_priority_init() is called to calculate priority following the below function:

$$\frac{W(p_i)}{\deg(p_i) + 1}$$

```
# randomly initialize the node strategies range in [0(out), 1(in)] (for requirements 1 only)
def node_strategy_init(G):
    strategy = {}
    for i in range(node_num):
        strategy[i] = random.randint(0, 1)
    nx.set_node_attributes(G, strategy, name = 'strategy')
```

Also, we need to initialize the strategy for each node. It can be either 0 or 1.

```python
# randomly pick up one player who can improve its utilty (weighted MIS game)
def node_choose1(G):
    players = []
    for i in range(node_num): # node i
        best_response = 1 # new best response to player i
        for j in G.neighbors(i): # neighbor node j
            if G.nodes[j]['priority'] >= G.nodes[i]['priority'] and G.nodes[j]['strategy'] == 1: # pj belogns to Li and cj = 1
                best_response = 0
                break
        if G.nodes[i]['strategy'] != best_response:
            players.append(i)
    if len(players) == 0:
        return -1
    return random.choice(players)
```

In each round of the game, we randomly pick one node which can improve its utility. To simplify the process, we can do it by following the best response, which is:

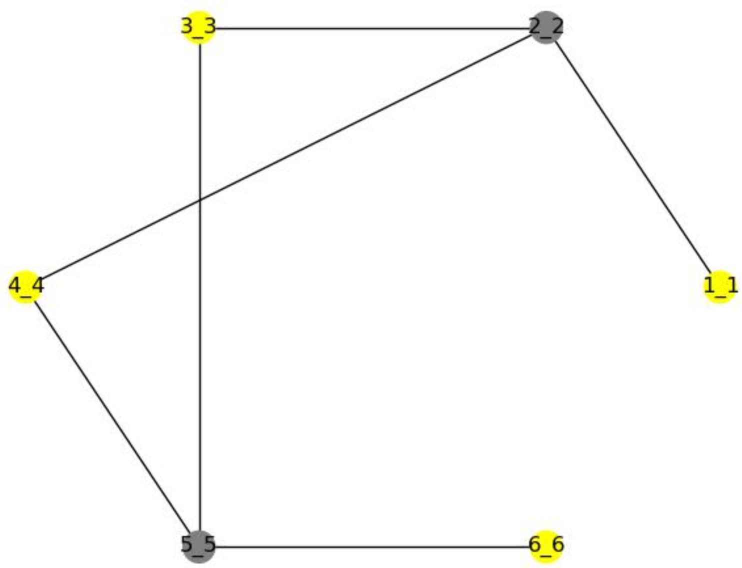
$$BR_i(c_{-i}) = \begin{cases} 0, & \text{if } \exists p_j \in L_i, c_j = 1 \\ 1, & \text{otherwise.} \end{cases}$$

The cardinality of this game should be the sum of all nodes who choose strategy 1(in).

In each game, we firstly initialize the game state. And then we repeatedly play the game until no one can further improve its utility. Note that to find out the extreme cardinality value in the game, we repeatedly play the weighted MIS game and record the maximum value.

```python
def weighted_MIS_game(graph):
    max_val = 0
    max_G = graph
    total_move_count = 0
    total_set_cardinality = 0
    for i in range(1000): # play 1000 times and observe the max cardinality value
        # initialize the graph and plot it
        G = graph
        node_weights_init(G)
        node_priority_init(G)
        node_strategy_init(G)

        move_count = 0
        node_count = 0
        while True:
            player_i = node_choose1(G)
            if player_i == -1:
                break
            G.nodes[player_i]['strategy'] = abs(1 - G.nodes[player_i]['strategy'])
            move_count += 1

        for n in range(node_num):
            if G.nodes[n]['strategy'] == 1:
                node_count += 1
        total_move_count += move_count
        total_set_cardinality += node_count
        if node_count > max_val:
            max_val = node_count
            max_G = G.copy()

    print("the cardinality of Weighted MIS Game is ", max_val)
```

Simulation Result



It is the result of the given test case.In this stimulated result, the format for the label of nodes is <index_weight>.
The result shows that it is a weighted MIS game. In this test case, the cardinality is 4.

Requirement 1-2: Symmetric MDS-based IDS Game

Code Explanation

The graph initialization, strategy initialization parts are the same as the previous game.

```
# randomly pick up one player who can improve its utilty (symmetric MDS-based IDS game)
def node_choose2(G):
    players = []
    for i in range(node_num):
        #check domination
        has_dominate = True
        M_i = [node for node in G.neighbors(i)] + [i] # closed neighbor
        for i_neighbor in M_i:
            v_j = 0
            for i_neighbor_neighbor in G.neighbors(i_neighbor):
                v_j += G.nodes[i_neighbor_neighbor]['strategy']
            if v_j == 0:
                has_dominate = False
                break
```

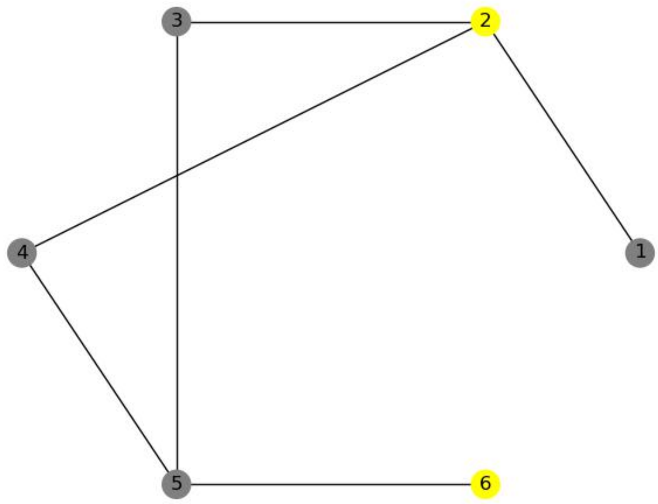
```
#check independence
not_independence = False
for neighbor_node in G.neighbors(i):
    if G.nodes[neighbor_node]['strategy'] == 1:
        not_independence = True
        break
```

To choose one node, we also follow the best response. From the given utility, we know that for the node to select strategy 1, two conditions are required: (i) there should not exist any of the closed neighbor nodes be dominated, (ii) all open neighbors of node i choose strategy 0.

The cardinality of this game should be the amount of the nodes who chooses strategy 1(in). To find out the extreme cardinality value in the game, we repeatedly play it and record the minimum value.

```
def symmetric_MDS_based_IDS_game(graph):
    min_val = len(graph.nodes())
    min_G = graph
    total_move_count = 0
    total_set_cardinality = 0

    for i in range(80000):
        move_count = 0
        node_count = 0

        G = graph
        node_strategy_init(G)
        while True:
            player_i = node_choose2(G)
            if player_i == -1:
                break
            G.nodes[player_i]['strategy'] = abs(1 - G.nodes[player_i]['strategy'])
            move_count += 1

        for n in range(node_num):
            if G.nodes[n]['strategy'] == 1:
                node_count += 1
        total_move_count += move_count
        total_set_cardinality += node_count
        if node_count < min_val:
            min_val = node_count
            min_G = G.copy()

    print("the cardinality of Symmetric MDS-based IDS Game is", min_val)
```

Simulation Result



The result shows that it is a symmetric MDS-based IDS game. And the initial state will affect the final result. Its cardinality is the minimum value of 2.

Requirement 2: Maximal Matching Game

Code Explanation

```
# randomly initialize the node strategies (N_i or null) (for requirements 2 only)
def node_strategy_init2(G):
    strategy = {}
    for i in range(node_num):
        profile = [n for n in G.neighbors(i)] + [-1] # open neighbors + unmatched
        strategy[i] = random.choice(profile)
    nx.set_node_attributes(G, strategy, name= 'strategy')
```

For a matching game, its strategy includes open neighbors and null. Each node randomly selects a strategy from one of them.

To solve this problem, we should define the utility function and best response.

The utility function:

$$U_i(C) = \begin{cases} \alpha & , \text{if } \exists P_j,\ C_i = P_j \text{ and } C_j = P_i \\ \beta & , \text{if } \exists P_j,\ C_i = P_j \text{ and } C_j \neq P_i \\ \gamma & , \text{if } \exists P_j,\ C_i = P_j \text{ and } C_j = \text{null} \\ 0 & , \text{if } C_i = \text{null} \end{cases}$$

where α > β > 0 > γ
In the utility function, we list all of the four cases and give it the corresponding payoff. The four cases are: (1) Forming a matching pair, (2) Forming a Mismatching pair, (3) Node_i points to the node choosing null, (4) Node_i chooses null.

We want to encourage the first two cases, so the utility is positive. The third case is not what we want, so we give them some penalty.

The best response:

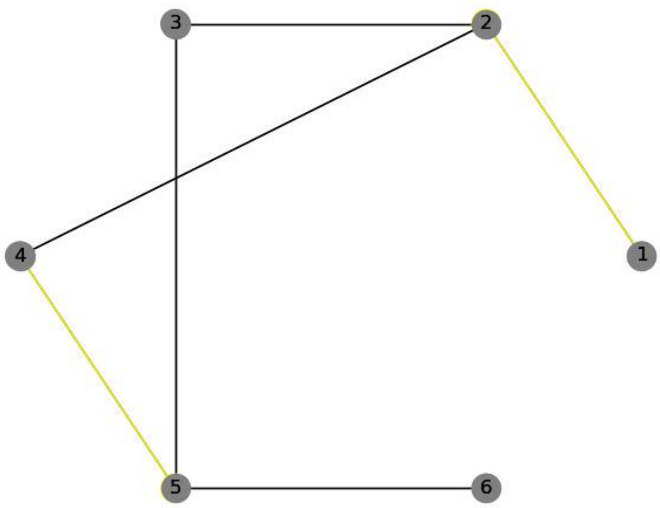
$$BR_i(C_{-i}) = \begin{cases} P_j & \text{if } \exists P_j \in N_i,\ C_i = P_j \text{ and } C_j = P_i \\ P_j & \text{if } \exists P_j \in N_i,\ C_i \neq P_j \text{ and } C_j = P_i \\ \text{random}(N_i) & \text{if } C_i = \text{null and } \forall P_j \in N_i,\ P_j = \text{null} \\ -1 & , \text{otherwise} \end{cases}$$

The best response should be: (1) when forming a matching pair, do not change anything, (2) If node_i chooses null but other node points it, node_i should points back, (3) if the closed neighbors of node_i all choose null, node_i should randomly choose one of its neighbors, (4) otherwise, node_i should choose null.

```python
def node_choose3(G):
    players_strategies = {}
    for i in range(node_num):
        point_to = G.nodes[i]['strategy'] # node i point to
        point_me = []  # nodes point to node i
        best_response = -1
        Ni_choose_null = [] # any of the neighbors choose null

        if point_to != -1 and i == G.nodes[point_to]['strategy']: # if pair matched, then do nothing
            continue

        for ni in G.neighbors(i):
            if G.nodes[ni]['strategy'] == i:
                point_me.append(ni)
            elif G.nodes[ni]['strategy'] == -1:
                Ni_choose_null.append(ni)

        if len(point_me) != 0: # if pair not matched yet and some neighbors point me, then randomly choose one
            best_response = random.choice(point_me)
        elif len(Ni_choose_null) != 0: # if pair not matched yet and no neighbor point me, then choose last one
            best_response = random.choice(Ni_choose_null)

        if best_response != point_to:
            players_strategies[i] = best_response
    if len(players_strategies) == 0:
        return -1, -1
    return random.choice(list(players_strategies.items()))
```

The cardinality of this game should be the amount of the edges for the matching pairs.

Simulation Result



The result shows it is a maximal matching game. And the initial state will affect the final result. In this case, the result cardinality will always be 2. But the matching edges may be different.

Note that in all of these three different games, it is played 1000, 8000 and 1000 times respectively to ensure the output value will be minimum or maximum of all the candidate answers. Therefore, the code takes some time to compute the final result.