Game Theory HW2
Graph Problem

Bo-Han Chen (陳柏翰)
Student ID:312551074
bhchen312551074.cs12@nycu.edu.tw

Experiment Environment

- OS: Windows 10 22H2
- Hardware: Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz
- Python 3.12.0
  - networkx 3.2.1
  - matplotlib 3.8.0

Requirement 1-1: Weighted MIS Game

Problem Statement

The goal of Weighted MIS Game is to maximize the total weight in the MIS set. In my implementation, the following function used to determine the priority of nodes is shown below:

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

For each player, the utility function is defined as:

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

where

$$\omega(c_i, c_j) = -\alpha c_i c_j, \alpha > 1$$

and

$$L_i = \{p_j | p_j \in N_i \wedge PR_i \leq PR_j\}$$

From the ultility function shown above, we can see the player will prefer to join the MIS set if there are no neighbor with higher priority in the set, which lead to the best response shown below:

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

Code Description

For the implementation of this assignment, I use *networkx* library's *Graph* structure to represent the graph. After parsing the command-line arguments, the constructor of *gameModel* will take the number of nodes, the edge list, the type of game and the times of simulation as input.

```
class gameModel():
    def __init__(self, num_node, link_list, game_type, iteration_time=1000):
        self.graph = nx.Graph()
        self.num_node = num_node
        self.link_list = link_list
        self.game_type = game_type
```

```
        self.iteration_time = iteration_time
```

Listing 1: Constructor of gameModel

Then the *init_graph* function will be called to initialize the nodes and edges of *Graph*.

```
def init_graph(self):
    h = nx.path_graph(self.num_node + 1)
    h.remove_node(0)
    self.graph.add_nodes_from(h)
    for src_node, link in enumerate(self.link_list):
        for des_node in range(len(link)):
            if link[des_node] == '1':
                self.graph.add_edge(src_node + 1, des_node + 1)
```

Listing 2: Graph Initialization

To initialize the weight, priority and strategy of nodes in weighted MIS game, the *init_node_MIS* function will be called.The weight is set as the node id, and the initial strategy of each node will be randomly selected from the strategy set $\{0, 1\}$.

```
def init_node_MIS(self):
    # initialize the weight of each node
    for node in self.graph.nodes():
        self.graph.nodes[node]['weight'] = node

    # initialize the priority of each node
    # first priority function
    for node in self.graph.nodes():
        neighbor_weight_sum = 0
        node_weight = self.graph.nodes[node]['weight']
        for neighbor in self.graph.neighbors(node):
            neighbor_weight_sum += self.graph.nodes[neighbor]['weight']
        priority = node_weight / neighbor_weight_sum + node_weight
        self.graph.nodes[node]['priority'] = priority

    # initialize the strategy of each node
    for node in self.graph.nodes():
        self.graph.nodes[node]['strategy'] = random.randint(0, 1)
```

Listing 3: Node Initialization of Weighted MIS Game

The simulation of Weighted MIS Game is shown in Listing 4. For each iteration, each node will be checked to see if it can get higher utility. If there are multiple nodes perfer to change their strategy, one of them will be randomly selected and change its strategy to best response. If there are no nodes will change their strategy, the simulation will be terminated since the NE is reached. The cardinality of the MIS set (the number of nodes in the MIS set) will be returned as the result.

```
def weighted_MIS(self):
    self.init_node_MIS()
    while True:
        # check if there are nodes can change the strategy
        candidate_nodes = []
        for node in self.graph.nodes():
            current_strategy = self.graph.nodes[node]['strategy']
            node_priority = self.graph.nodes[node]['priority']
            br = 1
            for neighbor in self.graph.neighbors(node):
                # only care about the nodes with higher or equal priority
                if self.graph.nodes[neighbor]['priority'] >= node_priority \
                    and self.graph.nodes[neighbor]['strategy'] == 1:
                    br = 0
                    break
            if current_strategy != br:
                candidate_nodes.append(node)
        # reach NE
        if len(candidate_nodes) == 0:
            break
```

```
        # randomly choose a node from candidate nodes
        node_to_change = random.choice(candidate_nodes)
        # change the strategy of the node
        self.graph.nodes[node_to_change]['strategy'] = 1 -
    self.graph.nodes[node_to_change]['strategy']
     cardinality = sum(nx.get_node_attributes(self.graph, 'strategy').values())

     return cardinality
```

Listing 4: Weighted MIS Game

Experiment Result

The graph of test case is shown in Figure 1:

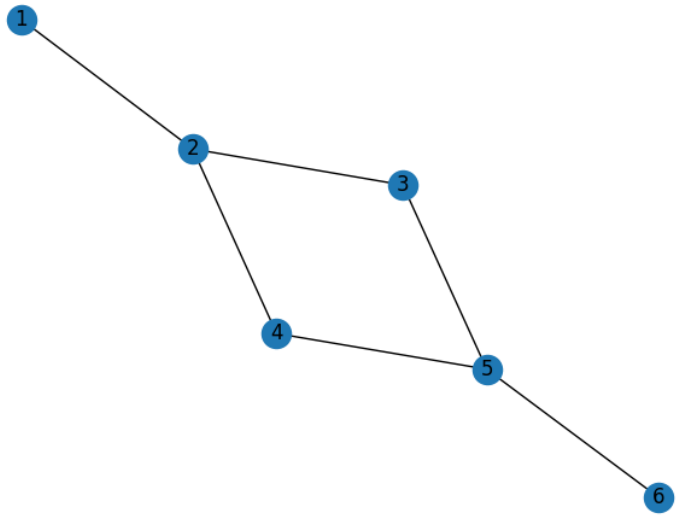

Figure 1: Test Case Graph

The result of Weighted MIS Game is shown in Figure 2, the node in the MIS set is colored in blue, and the red one is not in the MIS set. With the test case, we can see the result is correct since the total weight of MIS set is the maximum value, 14.

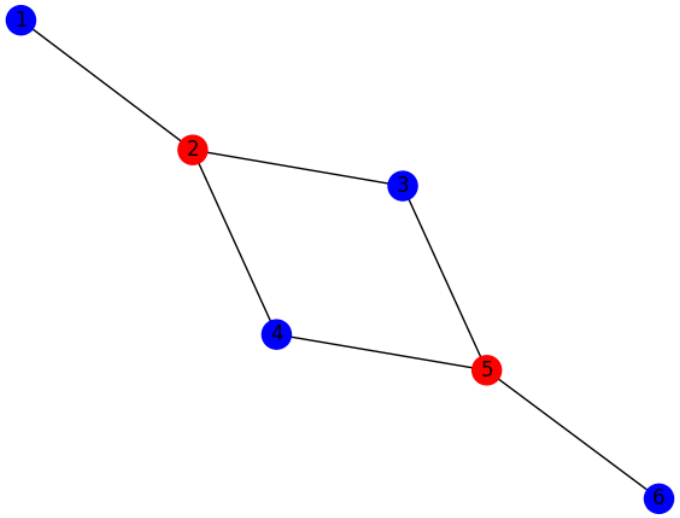

Figure 2: Result of Weighted MIS Game

Requirement 1-2: Symmetric MDS-based IDS Game

Problem Statement

By the definition of utility function in the slides:

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

We can know that the player will prefer to join the IDS set only if all following conditions are satisfied:

1. Avoid violating the independence rule: All of its' neighbors are not in the IDS set (the player itself is not dominated).
2. Gain of dominance: At least one of its' neighbors (p_j) is not dominated by players in set $N_j \cup \{p_j\}$.

So the best response of each player is shown below:

$$BR_i(C_{-i}) = \begin{cases} 0, & \text{if } \left(\exists p_j \in N_i, c_j = 1\right) \text{ or } \left(\forall p_j \in N_i, v_j(c_{-i}) \geq 1 \text{ and } \sum_{p_j \in N_i} c_j \geq 1\right) \\ 1, & \text{otherwise.} \end{cases}$$

where

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

Code Description

The simulation of Symmetric MDS-based IDS Game is shown in Listing 5. Like the implementation of previous section, nodes that can get higher utility will be randomly selected and change their strategy. First the independence rule will be checked to see if there is any neighbor is already in the IDS set. If so, the node will choose to leave the IDS set to avoid the panelty of violating the rule. Then we check whether the node itself is dominated by any of its' neighbors, why we not check the strategy of node itself is because the node will choose to leave the IDS set even if it is only dominated by itself, which leads to the wrong result. After checking the dominance of neighbors, if the node and all of its' neighbors are already dominated, the node will choose to leave the IDS set since it can not get any gain of dominance (which leads to $-\beta$ utility).
When reaching NE, the cardinality of IDS (number of nodes in IDS set) will be returned.

```
def symmetric_MDS_based_IDS(self):
    # initialize the strategy of each node
    for node in self.graph.nodes():
        self.graph.nodes[node]['strategy'] = random.randint(0, 1)

    while True:
        # check if there are nodes can change the strategy
        candidate_nodes = []
        for node in self.graph.nodes():
            current_strategy = self.graph.nodes[node]['strategy']

            br = 1
            independence_flag = 0
            domination_flag = 1
            v = 0
            # check the independence of the node
            for neighbor in self.graph.neighbors(node):
                if self.graph.nodes[neighbor]['strategy'] == 1:
                    independence_flag = 1
```

```
                    break
                v += self.graph.nodes[neighbor]['strategy']
            if independence_flag == 1:
                br = 0
            else:
                # check the neighbors are already dominated or not
                if v == 0:
                    domination_flag = 0
                else:
                    for neighbor in self.graph.neighbors(node):
                        v = self.graph.nodes[neighbor]['strategy']
                        for neighbor_neighbor in
self.graph.neighbors(neighbor):
                            v +=
self.graph.nodes[neighbor_neighbor]['strategy']
                        if v == 0:
                            domination_flag = 0
                            break
                if domination_flag:
                    br = 0

            if current_strategy != br:
                candidate_nodes.append(node)
        # reach NE
        if len(candidate_nodes) == 0:
            break
        # randomly choose a node from candidate nodes
        node_to_change = random.choice(candidate_nodes)
        # change the strategy of the node
        self.graph.nodes[node_to_change]['strategy'] = 1 -
self.graph.nodes[node_to_change]['strategy']
    cardinality = sum(nx.get_node_attributes(self.graph, 'strategy').values())

    return cardinality
```

Listing 5: Symmetric MDS-based IDS Game

Result and Discussion

For the provided test case, the cardinality of IDS set might be 2 or 4. The result is shown in Figure 3.

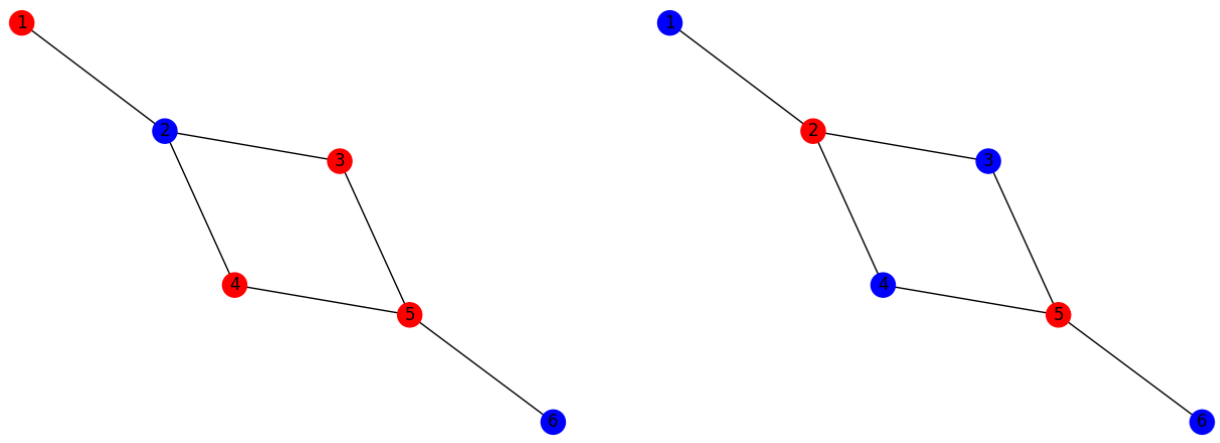

Figure 3: Result of Symmetric MDS-based IDS Game

We can see that some result of Symmetric MDS-based IDS Game is not the best NE, the reason is that the simulation is initialized with random strategy, and for each iteration, the node that can change the strategy is selected randomly. To ensure the result of Symmetric MDS-based IDS Game is the best NE, I set the running times of simulation to 300000 and record the minimum cardinality of IDS set.

Requirement 2: Matching Game

Problem Statement

In the matching game, each node will find the neighbor that is unmatched form a pair. In order to increase the number of matched pairs, each node will be assigned a priority based on its degree initially, and the node with higher priority will be prefered to form a pair. For the matching game, the utility function of each player can be defined as (H_i represents the set of higher priority neighbors, and L_i represents the set of lower priority neighbors):

$$u_i(C) = \begin{cases} \alpha, & \text{if } c_i = p_j \text{ and } c_j = p_i, p_j \in H_i \\ \beta, & \text{if } c_i = p_j \text{ and } c_j = null, p_j \in H_i \\ \gamma, & \text{if } c_i = p_j \text{ and } c_j = p_i, p_j \in L_i \\ \delta, & \text{if } c_i = p_j \text{ and } c_j = null, p_j \in L_i \\ \varepsilon, & \text{if } c_i = null \\ 0, & \text{if } c_i = p_j \text{ and } c_j = p_k, p_k \in N_j \setminus \{p_i, null\} \end{cases}$$

where

$$\alpha > \beta > \gamma > \delta > \varepsilon > 0$$

and

$$H_i = \{p_j | p_j \in N_i \wedge PR_j > PR_i\}$$
$$L_i = \{p_j | p_j \in N_i \wedge PR_j \leq PR_i\}$$
$$PR_i = \frac{1}{deg(p_i)}$$

With the utility function above, we can define the best response of each player as(HM_i represents the set of higher priority neighbors that match the node, HN_i represents the set of higher priority neighbors that not choose any nodes, LM_i represents the set of lower priority neighbors that match the node, LN_i represents the set of lower priority neighbors that not choose any nodes):

$$BR_i(C_{-i}) = \begin{cases} p_j, & p_j \in HM_i, \text{if } HM_i \neq \emptyset \\ p_k, & p_k \in HN_i, \text{if } HM_i = \emptyset \wedge HN_i \neq \emptyset \\ p_l, & p_l \in LM_i, \text{if } HM_i \cup HN_i = \emptyset \wedge LM_i \neq \emptyset \\ p_m, & p_m \in LN_i, \text{if } HM_i \cup HN_i \cup LM_i = \emptyset \wedge LN_i \neq \emptyset \\ null, & \text{otherwise.} \end{cases}$$

where

$$HM_i = \{p_j | p_j \in H_i \wedge c_j = p_i\}$$
$$HN_i = \{p_j | p_j \in H_i \wedge c_j = null\}$$
$$LM_i = \{p_j | p_j \in L_i \wedge c_j = p_i\}$$
$$LN_i = \{p_j | p_j \in L_i \wedge c_j = null\}$$

If there are neighbor with higher priority tend to form a pair with the node, the node will first choose it to form a pair, if there are no such neighbor, the node will choose the high-priority neighbor that are not choosing any node. If the node cannot match any high-priority neighbor, it will try to match the low-priority neighbor with the same condition mentioned above. Finally, if the node cannot match any neighbor, it will choose to be unmatched.

Code Description

The simulation of Matching Game is shown in Listing 6. Before the simulation, the strategy of each node will be randomly selected from the set including its' neighbors and 0 (unmatched), and the priority of each node will be calculated based on its degree. For each iteration, each node will be checked to see if it can get higher utility. If there are nodes perfer to change their strategy, *candidate_nodes* will store two list including the nodes and its best response. Then one of the nodes will be randomly selected and change its strategy to best response. Finally, the cardinality of the matched pairs will be returned as the result.

```
def matching_game(self):
        # initialize the strategy of each node
        # define null as 0
        for node in self.graph.nodes():
            strategy_set = list(self.graph.neighbors(node))
            strategy_set.append(0)
            # self.graph.nodes[node]['strategy'] = random.choice(strategy_set)
            self.graph.nodes[node]['strategy'] = 0

        # set up the priority of each node
        for node in self.graph.nodes():
            self.graph.nodes[node]['priority'] = 1 / self.graph.degree(node)

        while True:
            # check if there are nodes can change the strategy
            candidate_nodes = [[], []]
            for node in self.graph.nodes():
                current_strategy = self.graph.nodes[node]['strategy']
                current_priority = self.graph.nodes[node]['priority']

                # if node is matched, the best response won't change
                # if current_strategy != 0 and
    self.graph.nodes[current_strategy]['strategy'] == node:
                #     continue
                # if not matched, find the best response
                low_match_neighbor = []
                high_match_neighbor = []
                for neighbor in self.graph.neighbors(node):
                    if self.graph.nodes[neighbor]['strategy'] == 0:
                        if self.graph.nodes[neighbor]['priority'] >=
    current_priority:
                            high_match_neighbor.append(neighbor)
                        else:
                            low_match_neighbor.append(neighbor)
                    elif self.graph.nodes[neighbor]['strategy'] == node:
                        if self.graph.nodes[neighbor]['priority'] >=
    current_priority:
                            high_match_neighbor = [neighbor]
                        else:
                            low_match_neighbor = [neighbor]
                        break
                if len(low_match_neighbor) == 0 and len(high_match_neighbor) ==
    0:
                    br = 0
                elif len(high_match_neighbor) != 0:
                    br = random.choice(high_match_neighbor)
                else:
                    br = random.choice(low_match_neighbor)
                if current_strategy != br:
                    candidate_nodes[0].append(node)
                    candidate_nodes[1].append(br)
            if len(candidate_nodes[0]) == 0:
                break
            # randomly choose a node from candidate nodes
```

```
            node_to_change_idx = random.randint(0, len(candidate_nodes[0]) - 1)
            node_to_change = candidate_nodes[0][node_to_change_idx]
            # change the strategy of the node
            self.graph.nodes[node_to_change]['strategy'] =
    candidate_nodes[1][node_to_change_idx]
        # calculate the matched pairs
        cardinality = 0
        for node in self.graph.nodes():
            node_strategy = self.graph.nodes[node]['strategy']
            if node_strategy != 0 and
    self.graph.nodes[node_strategy]['strategy'] == node:
                cardinality += 1
        cardinality  = int(cardinality / 2)

        return cardinality
```

Listing 6: Matching Game

Result and Discussion

The result of Matching Game simulated on the provided test case is shown in Figure 4.

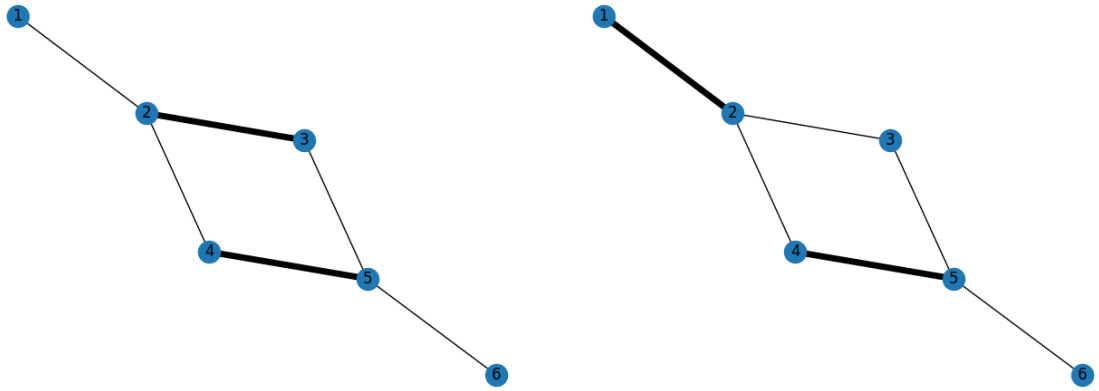

Figure 4: Some Result of Matching Game

For the test case, the cardinality of Matching Game is always the maximal and maximum, 2. To verify if there are any situation that the cardinality of simulation is not maximum, I try another test case shown in Figure 5.

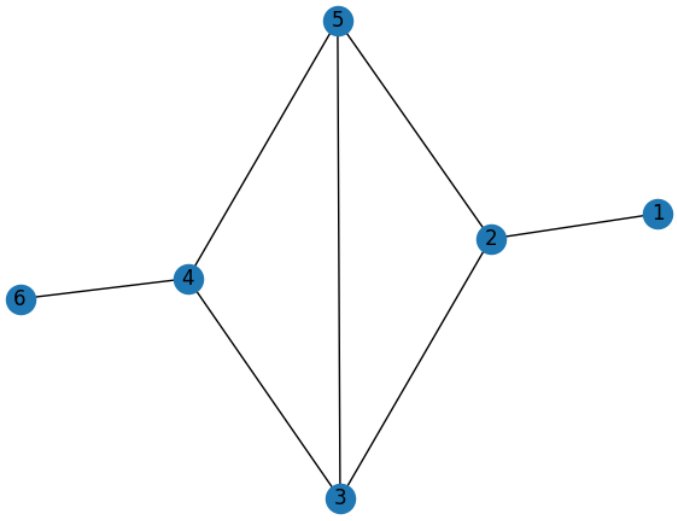

Figure 5: Test Case 2 for Matching Game

The result of Matching Game is shown in Figure 6.

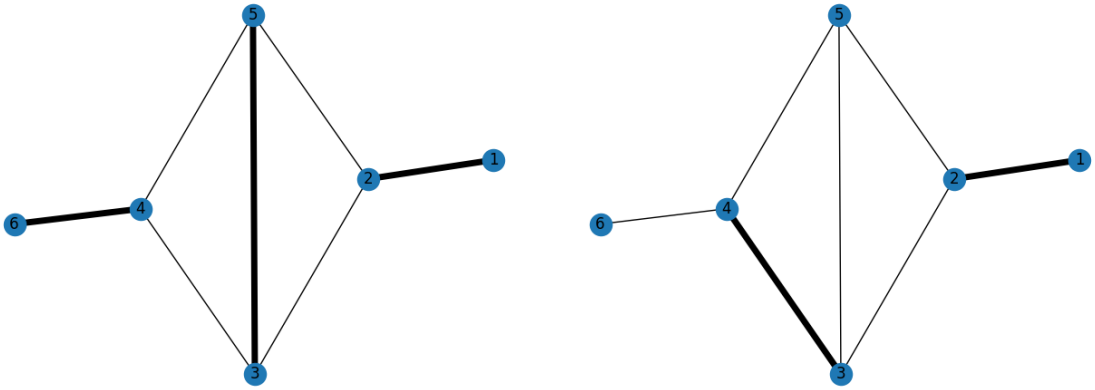

Figure 6: Result of Matching Game 2

As we can see, the cardinality of Matching Game is not always the maximum, for the result on the right, the cardinality is 2, which is maximal but not maximum. The reason is the same as mensioned in the previous section, with the random strategy initialization and random selection of nodes to change strategy, the result of simulation is not always the best NE. In Matching Game simulation, I set the running times of simulation to 10000 get record the maximum cardinality to ensure the result is the best NE.