Assignment II: Graph Problem

Code Description

base_model

```
import numpy as np
from abc import ABC, abstractmethod

class base_model(ABC):
    def __init__(self, n, connects):
        self.n = int(n)
        self.G = self.init_connection(connects)
        self.state = self.init_state()
        self.deg = self.init_deg()

    def init_connection(self, connects):
        G = [[] for _ in range(self.n)]
        for i in range(self.n):
            for j in range(len(connects[i])):
                if connects[i][j] == '1':
                    G[i].append(j)
        return G

    def init_deg(self):
        deg = np.zeros(self.n, dtype=np.int_)
        for i in range(self.n):
            deg[i] = len(self.G[i])
        return deg

    def init_state(self):
        state = np.zeros(self.n, dtype=np.bool_)
        for i in range(self.n):
            state[i] = np.random.randint(0, 2)
        return state

    @abstractmethod
    def utility(self, player):
        pass

    @abstractmethod
    def best_response(self, player):
        pass

    def reach_NE(self):
        can_improve = []
        for i in range(self.n):
            if self.state[i] != self.best_response(i):
                can_improve.append(i)
        return can_improve

    def solve(self):
        move_count = 0
        while True:
            can_improve = self.reach_NE()
            if can_improve == []:
                break
            player = np.random.choice(can_improve)
            self.state[player] = self.best_response(player)
            move_count += 1
        return move_count, np.sum(self.state)
```

- 由於三個任務間對於以下內容皆類似，因此我撰寫一個base_model來實作以下功能
  - 圖的建構 (init_connection)
    - 利用adjacency list建構整張圖
  - degree的計算 (init_deg)
    - 計算每個player的out degree(=in degree)
  - state初始化方式 (init_state)
    - 隨機初始化一個boolean list，紀錄當前每個player的選擇情況(0/1)
    - 另外，由於在Mactching Game中每個player的選擇並非只有0/1，因此在Matching Game中會需要override此function
  - 是否已達到Nash Equilibrium (reach_NE)
    - 確認每個player是否無法透過改變自身state來增加utility
    - 將有辦法增加utility的所有player回傳，否則回傳一個空的list
  - 模擬整個Game Process (solve)
    - 每次從還可以增加utility的那些player中隨機選一個並改變state為其best response
    - 直到沒有任何player可以增加utility時回傳結果
- 另外，由於不同game的utility與best response可能不太相同，因此這兩個function在不同game中會另外實作。

1-1.MIS-based IDS Game

```
import numpy as np
from base_model import base_model

class mis_based_ids_model(base_model):
    def __init__(self, n, connects):
        super().__init__(n, connects)
        self.alpha = 1.5

    def utility(self, player):
        if self.state[player] == 0:
            return 0

        u = 1
        for other in self.G[player]:
            if self.deg[player] <= self.deg[other]:
                u -= self.alpha * self.state[other]
        return u

    def best_response(self, player):
        for other in self.G[player]:
            if self.state[other] == 1 and self.deg[player] <= self.deg[other]:
                return 0
        return 1
```

- 在 MIS-based IDS Game中需額外實作兩個Funtion
  - utility
    - 若當前player的state為0
      - 回傳0
    - 若當前player的state為1
      - let u = 1
      - 計算u - α(=1.5) * 所有neighbors(other)中deg $\geq$ 當前player且state為1的個數
      - 回傳u
  - best_response
    - 若當前player的neighbors(other)中有任一deg $\geq$ 當前player並且其state為1
      - 回傳此player的BR為0
    - 反之
      - 回傳此player的BR為1

1-2.Symmetric MDS-based IDS Game

```
from base_model import base_model

class symmetric_mds_based_ids_model(base_model):
    def __init__(self, n, connects):
        super().__init__(n, connects)
        self.alpha = 1.5
        self.beta = 0.5
        self.gamma = self.n * self.alpha + 1

    def gain(self, player):
        v = self.state[player]
        for other in self.G[player]:
            v += self.state[other]
        if v == 1:
            return self.alpha
        return 0

    # gain of dominance
    def total_gain(self, player):
        g = self.gain(player)
        for other in self.G[player]:
            g += self.gain(other)
        return g

    # penalty of violating independence
    def penalty(self, player):
        w = 0
        for other in self.G[player]:
            w += self.state[player] * self.state[other] * self.gamma
        return w

    def utility(self, player):
        if self.state[player] == 0:
            return 0
        return self.total_gain(player) - self.beta - self.penalty(player)

    def best_response(self, player):
        orginal_state = self.state[player]
        orginal_state_utility = self.utility(player)
        other_state = 1 - orginal_state
        self.state[player] = other_state
        other_state_utility = self.utility(player)
        self.state[player] = orginal_state
        if other_state_utility > orginal_state_utility:
            return other_state
        return orginal_state
```

- 在 MIS-based IDS Game中需實作五個Funtion
  - gain
    - 此function為計算輸入的player中是否其與其neighbors的state總和為1
      - 若為1，代表此player剛好是dominated別人或剛好只有一個neighbor dominated此player
        - 回傳α(=1.5)
      - 反之
        - 回傳0
  - total_gain
    - 此function為計算player與其neighbors的gain總和
  - penalty
    - 此function為計算player與其neighbor是否有違反independence
      - 若player與其neighbor兩者state同時為1，則累加一個$\gamma(=\#players * \alpha + 1)$
      - 回傳其總和
  - utility
    - 若當前的player的state為0
      - 回傳0
    - 若當前的player的state為1
      - 回傳total gain - β(=0.5) + penalty

2.Matching Game

```
import numpy as np
from base_model import base_model

class matching_model(base_model):
    def __init__(self, n, connects, heuristic=True):
        super().__init__(n, connects)
        self.state = self.init_state()
        self.prefer_table = self.init_prefer_table(heuristic)
        self.G = self.add_self_loop(self.G)
        self.beta = 0.5

    def add_self_loop(self, G):
        for i in range(self.n):
            self.G[i].append(i)
            self.G[i] = sorted(self.G[i])
        return G

    def init_state(self):
        state = np.zeros(self.n, dtype=np.int_)
        for i in range(self.n):
            state[i] = np.random.choice(self.G[i])
        return state

    def init_prefer_table(self, heuristic):
        prefer_table = np.zeros((self.n, self.n), dtype=np.int_)
        for i in range(self.n):
            for j in range(self.n):
                if i == j:
                    prefer_table[i][j] = 0
                elif j in self.G[i]:
                    if heuristic == True:
                        prefer_table[i][j] = self.n - self.deg[j]
                    else:
                        prefer_table[i][j] = 1
        return prefer_table

    def utility(self, player):
        if self.state[player] == player:
            return 0
        else:
            u = 0
            chosen = self.state[player]
            prefer_rank = self.prefer_table[player][chosen]
            can_link = self.state[chosen] == player
            can_link |= self.state[chosen] == chosen
            u = prefer_rank * can_link - self.beta
        return u

    def best_response(self, player):
        origin_state = self.state[player]
        return_state = self.state[player]
        max_state_utility = self.utility(player)
        others = self.G[player]
        for other in others:
            self.state[player] = other
            other_state_utility = self.utility(player)
            if other_state_utility > max_state_utility:
                return_state = other
                max_state_utility = other_state_utility
        self.state[player] = origin_state
        return return_state

    def make_pair(self):
        matching = []
        for i in range(self.n):
            for other in self.G[i]:
                if i < other and self.state[i] == other and self.state[other]
                    matching.append((i, other))
        return len(matching)

    def solve(self):
        move_count = 0
        while True:
```

```
            can_improve = self.reach_NE()
            if can_improve == []:
                break
            player = np.random.choice(can_improve)
            self.state[player] = self.best_response(player)
            move_count += 1
        return move_count, self.make_pair()
```

- 在 Matching Game中需實作七個Funtion
  - add_self_loop
    - 由於在matching game中會有player沒有與其他player配對，因此需要加上self loop作為unmatched的情況
  - init_state
    - 與1-1, 1-2不同，player並非只有0/1的選擇，因此在matching game的初始化中，將所有player指向其任一neighbors(包括自己)
  - init_prefer_table
    - 此部分可以依照是否使用heuristic來調整
      - 若沒有使用heuristic，則prefer_table(i, j): 代表player i連到 player j的priority，共分為三種情況
        - $i = j$
          - 0
        - $i \neq j$ and i and j are neighbors
          - 1
    - 倘若為使用heuristic，則只需更改以下部分
      - $i \neq j$ and i and j are neighbors
        - #players - degree of player j
  - utility
    - 若當前player的state為自己，代表沒有與他人產生pair
      - 回傳0
    - 反之
      - 可能產生連線共有兩個情況
        - 對方(chosen)連向當前的player，可以形成matching
        - 對方(chosen)連向他自己(self loop)，當前player指向他並不會破壞任何已產生的matching
      - 計算出對於當前player與其指向的player(chosen)的priority
      - 回傳priority(prefer_rank) * 是否可以產生連線 - β(=0.5)
  - best_response
    - 回傳對於當前player中，模擬其連向其他neighbors(包括自己)是否可以產生更好的utility
      - 若可以產生更好的utility
        - 回傳可以產生最大的utility時所連向的neighbor
      - 反之
        - 回傳原始連向的neighbor
  - make_pair
    - 計算達到NE時的state中產生matching的數量
      - 對於產生的matching有幾個設定條件，對於任意player i, j 而言
        - i < j(確保沒有self loop與連線被計算兩次)
        - state[i] ＝ j and state[j] = i(兩者互相連線)
      - 回傳產生matching的數量
  - solve
    - 與base model的solve大致相同，僅差別於計算完state後需要將state轉換成產生的matching數量(make_pair)後再回傳。

Result Analysis

Experimental Setups

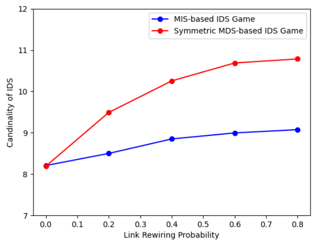
- WS model
  - 對於requirement 1-1, 1-2, 2，皆使用以下列參數設計，並比較不同rewiring probability對於模型的影響。
    - number of nodes
      - 30
    - number of edges
      - 4
    - rewiring Probability
      - [0.0, 0.2, 0.4, 0.6, 0.8] (如下圖所示)
      - 
  - 每組rewiring probability皆產生出1000組新的WS Model，分別計算在requirement 1-1, 1-2, 2的Cardinality平均與標準差。

Requirement 1-1, 1-2

- 實驗結果

Rewiring_Probs	MIS-based IDS Game	Symmetric MDS-based IDS Game
0.0	8.207 ± 0.623	8.191 ± 0.636
0.2	8.5 ± 0.883	9.491 ± 0.959
0.4	8.85 ± 1.075	10.254 ± 1.091
0.6	8.996 ± 1.105	10.688 ± 1.165
0.8	9.075 ± 1.132	10.785 ± 1.210

- 實驗圖表



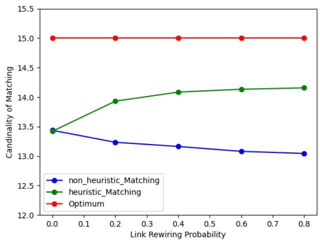
- 實驗分析
  - 從表格中可以發現，將rewiring probability增加，皆會使Requirement 1-1與 Requirement1-2的mean與std增加。
  - 而從圖表中可以發現，MIS-based IDS model的結果相較於Symmetric MDS-based IDS model好上不少，我認為是因為前者只選擇有更大或相同degree的neighbors，而後者並無此限制，因此若後者改為Asymmetric應該能使結果更好。

Requirement 2

- 實驗結果

Rewiring_Probs	non_Heuristic_Matching	Heuristic_Matching	Optimum
0.0	13.437 ± 0.668	13.420 ± 0.663	15 ± 0
0.2	13.234 ± 0.718	13.931 ± 0.615	15 ± 0
0.4	13.163 ± 0.754	14.085 ± 0.561	15 ± 0
0.6	13.08 ± 0.791	14.133 ± 0.558	15 ± 0
0.8	13.044 ± 0.832	14.157 ± 0.571	15 ± 0

- 實驗圖表



- 實驗分析
  - 從表格中可以發現，對於non_heuristic_Matching而言，將rewiring_probs升高，則會讓結果變差，而對於heuristic_Matching而言則相反。我認為其原因是因為對於rewiring_prob為0的情況，是否有heuristic對於此問題完全沒有任何影響，因為所有的degree皆為4。
  - 而將rewiring_probs升高，會使degree改變，進而使heuristic_Matching產生較大的作用，因此heuristic_Matching的結果會更好，而從圖表我們同樣也能看到此現象。