Project Structure

The codes of my hw1 is hosted on Github
The project structure is as follows

```
310551055_HW1
├── game_solver.py
├── graph_games.py
├── graph.py
├── graphviz.ipynb
├── 310551055_code.py
├── 310551055_report.pdf
└── README.md
```

Execute Code

To run my code use the following command.
The text experiment results of Part1 and Part2 will be printed.

```shell
cd 310551055_HW1
python3 310551055.py
```

Visulization

Also, you can see visualization of graph and plots in this report in `graphviz.ipynb` github notebook link
In order to run visualization , you need to install graphviz

```shell
pip install graphviz
```

Part 1

Graph Data

The graph data structure is defined in `graph.py`

- Each graph node is represented by a python string.
- We store all graph nodes in a set of string `Graph.nodes: Set[str]`
- We store all edges in a dictionary mapping from nodes to set of nodes `Graph.edges: Dict[str, [Set[str]]`

```python
class Graph:
    def __init__(self) -> None:
        # map each node index to the set of its neibours' indexes
        self.edges: Dict[str, Set[str]] = defaultdict(set)
        self.nodes: Set[str] = set()

    def node(self, node: str) -> None:
        self.nodes.add(node)

    def edge(self, node1: str, node2: str) -> None:
        self.edges[node1].add(node2)
        self.edges[node2].add(node1)

    def edgeDel(self, node1: str, node2: str) -> None:
        self.edges[node1].remove(node2)
        self.edges[node2].remove(node1)

    def nodeDel(self, node) -> None:
        if node in self.nodes:
            self.nodes.remove(node)
            for n in self.edges[node]:
                self.edges[n].remove(node)
            del self.edges[node]

    def addEdges(self, pairs: Iterable[Tuple[str, str]]) -> None:
        map(lambda n1, n2: self.edge(n1, n2), pairs)

    def neighbors(self, node: str) -> None:
        return self.edges[node]

    def degree(self, node: str) -> None:
        return len(self.edges[node])

    def clone(self) -> "Graph":
        return copy.deepcopy(self)
```

Watts-Strogatz Graph Initialization

Initialization of WS Graph is also defined in `graph.py`
To initialize the graph with WS model , we following the definition in Wikipedia:

> Watts–Strogatz graph
>
> Given the desired number of nodes N, the mean degree K (assumed to be an even integer), and a parameter β, all satisfying $0 \le \beta \le 1$ and $N \gg K \gg \ln N \gg 1$, the model constructs an undirected graph with N nodes and $\frac{NK}{2}$ edges in the following way:
>
> 1. Construct a regular ring lattice, a graph with N nodes each connected to K neighbors, $K/2$ on each side. That is, if the nodes are labeled $0 \ldots N-1$, there is an edge (i, j) if and only if $0 < |i-j| \bmod \left(N-1-\frac{K}{2}\right) \le \frac{K}{2}$.
> 2. For every node $i = 0, \ldots, N-1$ take every edge connecting i to its $K/2$ rightmost neighbors, that is every edge $(i, j \bmod N)$ with $i < j \le i + K/2$, and rewire it with probability β. Rewiring is done by replacing $(i, j \bmod N)$ with (i, k) where k is chosen uniformly at random from all possible nodes while avoiding self-loops ($k \ne i$) and link duplication (there is no edge (i, k') with $k' = k$ at this point in the algorithm).

Sepecifically, in the **link rewiring** part, if a link is to be rewried, we will randomly link current node to some random other node which is not yet linked to curerent node.

```
def randomWSGraph(n=16, k=4, link_rewiring_prob=0.0):
    """randomly initialize a graph using Watts-Strogatz Model"""
    # k should be even
    assert k % 2 == 0
    ws = Graph()

    for i in range(n):
        ws.node(str(i))

    for i in range(n):
        for dk in range(1, k//2+1):
            j = (i + dk) % n
            if random.random() < link_rewiring_prob:
                # change j to a random node that is not already connected to i
                j = random.choice(
                    list(n for n in ws.nodes if n not in ws.neighbors(str(i)) and n != str(i)))
            ws.edge(str(i), str(j))
    return ws
```

Game Definition

The definition of a finite strategy game is defined in `graph_games.py`

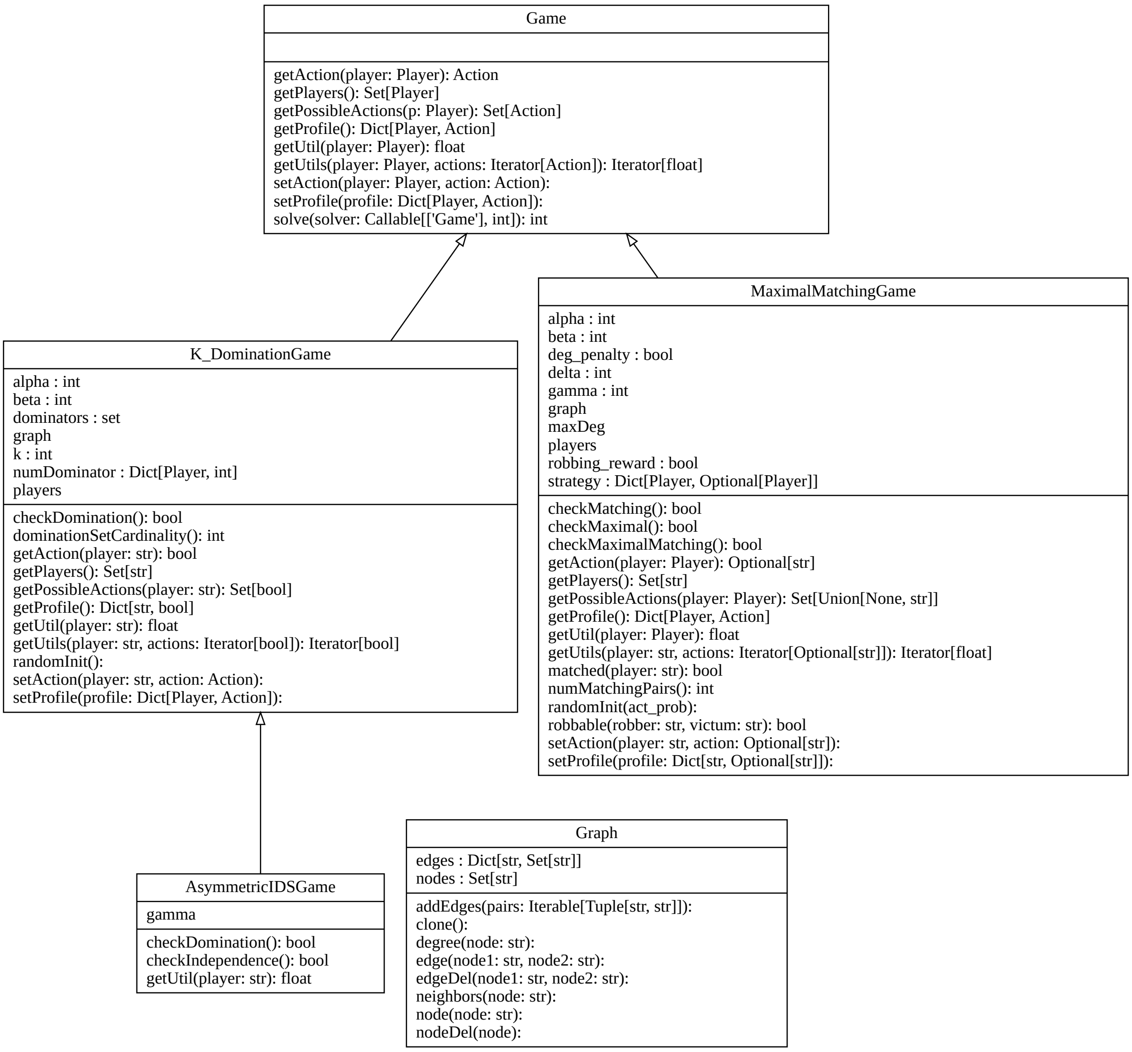
- Every graph games in the following Part1.1, Part1.2, Part2 are all instance to `Game` class.
- A Game Object should maintain its own internal state, and implement the following functions as interface for
  1. query game information (`getPlayers`, `getAction`, `getProfile`, `getPossibleActions`)
  2. query utility function (`getUtil`, `getUtils`)
  3. modify player stategy (`setAction`, `setProfile`)

```
class Game:
    """
        A Game Object should maintain its own internal state,
        and implement the following functions as interface for
            1. query game information ( getPlayers, getAction, getProfile, getPossibleActions )
            2. query utility function ( getUtil, getUtils )
            3. modify player stategy  ( setAction, setProfile)

    """

    def __init__(self) -> None:
        return

    def getPlayers(self) -> Set[Player]:
        """ return the set of all players in the game"""
        pass

    def getAction(self, player: Player) -> Action:
        """ return current action of a player"""
        pass

    def getProfile(self) -> Dict[Player, Action]:
        """ return strategy profile for current game state (action for each player) """
        pass

    def getPossibleActions(self, p: Player) -> Set[Action]:
        """ return all posible actions for a single player"""
        pass

    def getUtil(self, player: Player) -> float:
        """
         Return the utilities of a player in current strategy profile
        """
        pass
```

```python
    def getUtils(self, player: Player, actions: Iterator[Action]) -> Iterator[float]:
        """
            Return the utilities of a player for a set possible actions
            Parameters
                player: the player whose utility to return
                actions: contains a iterator of possible actions for current player
            Return
                a list of scores corresponding to actions
        """
        pass

    def setAction(self, player: Player, action: Action) -> None:
        """ set the current strategy of a player to certain action"""
        pass

    def setProfile(self, profile: Dict[Player, Action]) -> None:
        """ set the current strategy profile to argument"""
        pass

    def solve(self, solver: Callable[["Game"], int]) -> int:
        """solve the game using the given solver, return the iterations used"""
        return solver(self)
```

UML About Derived Classes of `Game`

In the diagram, we can see the `K_DominationGame` in Part 1.1, `AsymmetricIDSGame` in Part 1.2, and `MaximalMatchingGame` in Part 2 are all inherited from `Game`



Solver Definition

Solver for `Game` class is defined in `game_solver.py`, only a single solver (based on best-responce path) is implemented in this homework.

Solve Single Step

The solver is implemented as a function, we first define how to solve a single step .

- In each single step, the solver test each player in `population` (in the order presented in the list), whether it is already in best responce action.
  - If we find the first player p in `population` that is not using best responce , update the player's move to a best responce action, then return the player that is updated.

- **If all player is currently in best response action, we know the game is in Nash Equilibrium, return** `None`

```python
def bestResponseSingleStep(g: Game, population: List[Player]) -> Optional[Player]:
    """
    solve for NE using best response path for single step
    Input:
        - g: an instance of Game in graph_game
        - population: a list containing all players in Game g, the algorithm will check
          possible update using best response in the given order
    Return:
        None or player:
          - if the return value is None, means g is already in Nash Equilibrium
          - if the return value is a player, means we updated the player in last step
    """
    for p in population:
        current_util = g.getUtil(p)
        possible_acts = list(g.getPossibleActions(p))
        possible_utils = g.getUtils(p, possible_acts)
        # the first occorrance of the best responce
        best_act_index, best_util = max(
            enumerate(possible_utils), key=operator.itemgetter(1))

        if current_util == best_util:
            continue
        else:
            best_act = possible_acts[best_act_index]
            g.setAction(p, best_act)
            return p
    return None
```

Solve Until Convergence

In the full solver, we iteratively solve for best responce until the game reaches NE.
Note that we **randomly shuffle the** `population` **between each iteration**, thus preventing bias in the solving process.

```python
def bestResponseSolver(g: Game) -> int:
    """
    Trace along best-response path toward a Nash-Equilibrium
    Input:
        A instance of Game defined in graph_games
    Return:
        number of iterations during the solving,
        each iteration we overwrite the action of a player with his best response.
    """
    population = list(g.getPlayers())
    for total_iters in itertools.count(start=1, step=1):
        # get random order of the population
        random.shuffle(population)
        if bestResponseSingleStep(g, population) == None:
            return total_iters
```

Part 1.1

k-Domination Game [YC14]

- Players: node set $\{p_1, p_2, \ldots, p_n\}$
- Strategies: {0 (OUT), 1 (IN)}

If $|N_i| < k$, c_i must be 1

$$u_i(C) = \begin{cases} \boxed{\alpha \qquad \text{if } |N_i| < k \text{ and } c_i = 1} \\ \sum_{p_j \in N_i} g_j(C) - \beta & \text{if } |N_i| \geq k \text{ and } c_i = 1 \\ 0 & \text{otherwise} \end{cases}$$

where

$$g_i(C) = \begin{cases} \alpha, & \text{if } c_i = 1 \text{ and } v_i(C) \leq k \\ 0, & \text{otherwise} \end{cases} \qquad \boxed{\alpha > \beta > 0}$$

where

$$v_i(C) = \sum_{p_j \in N_i} c_j$$

N_i (not M_i): p_i's open neighbors (p_i excluded)

```python
class K_DominationGame(Game):
    """
    Simulate a K-Domination Game
    """

    def __init__(self, k: int,  graph: graph.Graph, alpha=2, beta=1) -> None:
        """
        make a K-domination Game from a given Graph
        Parameter:
            k: the minimum number of dominator-neighbors required
                 for each non-dominator node in a valid solution
            graph: an instance of Graph from graph.py
            alpha: utility gain for a player choosing "True"
                when a neibouring node is not yet k-dominated.
            beta: utility penalty (cost) for a player choosing "True"
        Note: we should have alpha > beta > 0 in order to
            get Nash Equilibriums that correspond to K-Dominating Sets
        """
        assert 0 < beta and beta < alpha
        # whether record whether a player is in the domination set
        self.k = k
        self.graph = graph
        self.dominators = set()
        self.players = self.graph.nodes
        self.alpha = alpha
        self.beta = beta
        # cache the number of nodes that dominate a particular node for faster
        # self.numDominator[p] should equal to
        #  len(set.intersection( graph.neibors, self.dominators ))
        self.numDominator: Dict[Player, int] = defaultdict(lambda: 0)

    def randomInit(self) -> None:
        """randomly initialize the strategies for each player"""
        self.dominators.clear()
        for p in self.players:
            if random.randint(0, 1) > 0:
                self.dominators.add(p)
                for n in self.graph.neighbors(p):
                    self.numDominator[n] += 1

    def getPlayers(self) -> Set[str]:
        return self.players.copy()
```

```python
    def getPossibleActions(self, player: str) -> Set[bool]:
        return set([True, False])

    def getAction(self, player: str) -> bool:
        return player in self.dominators

    def getProfile(self) -> Dict[str, bool]:
        profile = defaultdict(lambda: False)
        for p in self.players:
            profile[p] = True
        return profile

    def getUtil(self, player: str) -> float:
        if player not in self.dominators:
            return 0
        elif self.graph.degree(player) < self.k:
            return self.alpha
        else:
            def g(n):
                if n not in self.dominators and self.numDominator[n] <= self.k:
                    return self.alpha
                else:
                    return 0
            return sum(g(n) for n in self.graph.neighbors(player)) - self.beta

    def getUtils(self, player: str, actions: Iterator[bool]) -> Iterator[bool]:
        original_action = player in self.dominators
        utils = []
        for a in actions:
            self.setAction(player, a)
            utils.append(self.getUtil(player))
        self.setAction(player, original_action)
        return utils
```

```python
    def setAction(self, player: str, action: Action) -> None:
        if player in self.dominators:
            if action == False:
                self.dominators.remove(player)
                for n in self.graph.neighbors(player):
                    self.numDominator[n] -= 1
        else:
            if action == True:
                self.dominators.add(player)
                for n in self.graph.neighbors(player):
                    self.numDominator[n] += 1

    def setProfile(self, profile: Dict[Player, Action]) -> None:
        for p in self.players:
            self.setAction(p, profile[p])

    def checkDomination(self) -> bool:
        """check whether K-Domination condition is met"""
        return all(self.numDominator[p] >= self.k for p in self.players-self.dominators)

    def dominationSetCardinality(self) -> int:
        assert self.checkDomination(
        ), "The game hasn't been solved yet, the current solution is not k-domination set"
        return len(self.dominators)
```

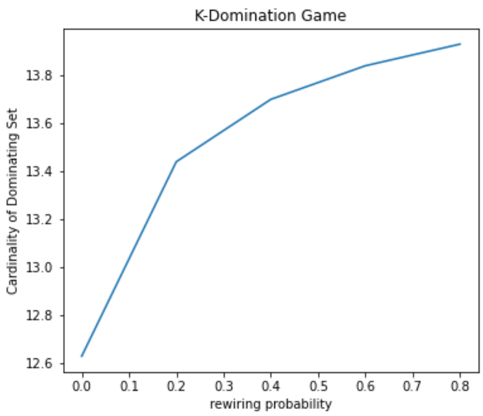
Run Experiments

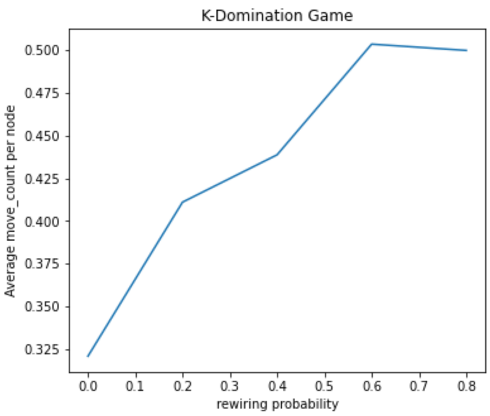
We run the experiments of both Part 1.1 and Part 1.2 together

```python
print("== Part 1" + "="*20)
for game in ["K_DominationGame", "AsymmectricIDSGame"]:
    print("-"*30)
    print(f'Game = {game}')
    print(f'{"rewiring_prob":15}, {"move_counts per node":15}, {"cardinality":15}')
    for rewire_prob_times_10 in range(0, 10, 2):
        rewiring_prob = rewire_prob_times_10 / 10
        move_counts = []
        cardinalities = []
        for i in range(100):
            g = graph.randomWSGraph(
                n=30, k=4, link_rewiring_prob=rewiring_prob)
            if game == "K_DominationGame":
                gg = K_DominationGame(2, g)  # run with k = 2
            else:
                gg = AsymmetricIDSGame(g)
            move_count = gg.solve(bestResponseSolver)
            cardinality = gg.dominationSetCardinality()
            move_counts.append(move_count)
            cardinalities.append(cardinality)
            if game == "K_DominationGame":
                assert gg.checkDomination()
            else:
                assert gg.checkDomination()
                assert gg.checkIndependence()
        print(
            f'{rewiring_prob:15.2f}, {sum(move_counts)/100 / 30:15.2f}, {sum(cardinalities)/100:15.2f}')
```

Averge over 100 runs, `rewiring_ probability`'s step-size = 0.2

This result is ploted using all parameters given in the spec.



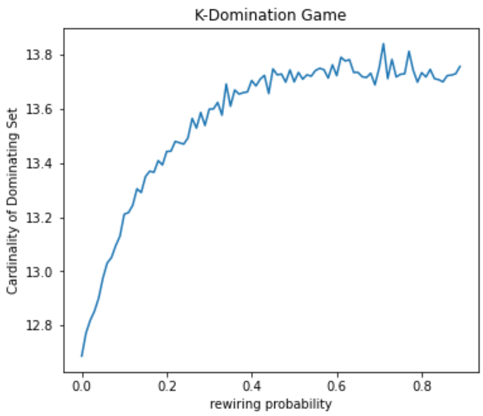


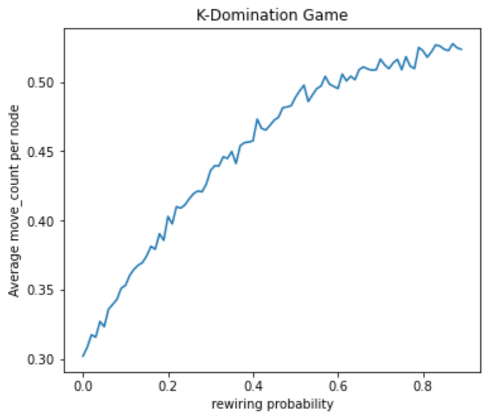
Average over 1000 runs, `rewiring_ probability`'s step-size = 0.01 (takes about 90 seconds on my laptop)

For more accurate visualization, we increase the number of runs to 1000, and shrink the step size to 0.01.
In the following diagram, we can see that

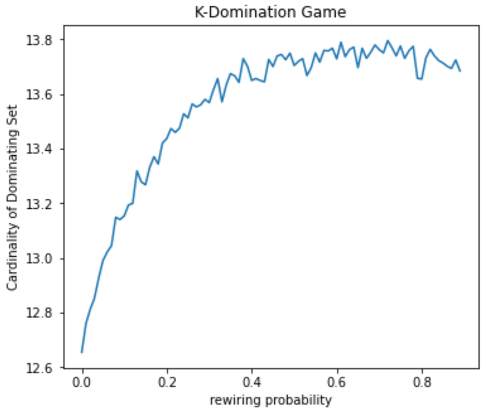
1. The game reach NE after more moves if we don't do initialization.
2. For K-Domination Game
    1. The variance of cardinality of game solution is high, even if we average over 1000 differnt runs.

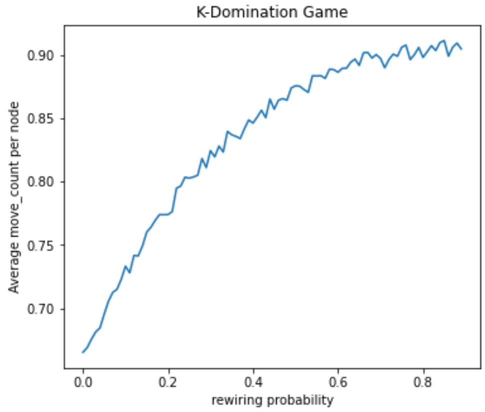
With Random Initialization





Without Random Initialization





Part 1.2

Definition of `AsymmetricIDSGame`

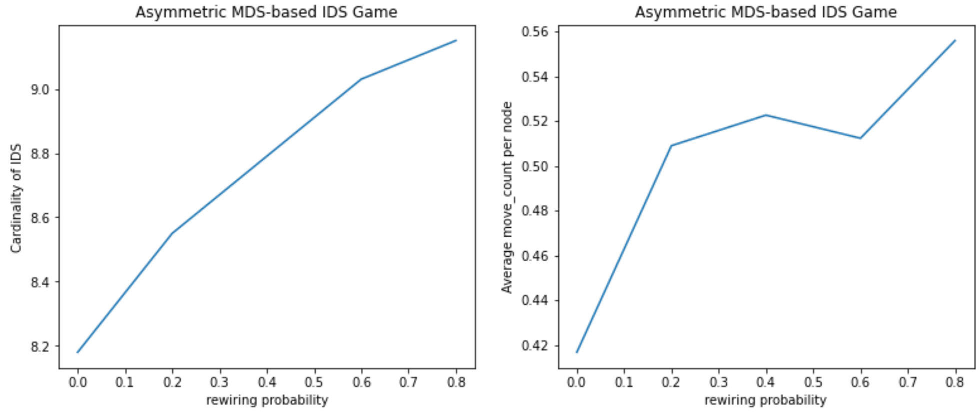
```python
class AsymmetricIDSGame(K_DominationGame):
    """
    Asymmectric Minimum-Dominating-Set-based Independent Dominateding Set Game

    we use K_DominationGame as base class and overwrite the following functions:
    - getUtil()
    - checkDomination()
    we also add a new function:
    - checkIndependence()

    """

    def __init__(self,  graph: graph.Graph, alpha=2, beta=1) -> None:
        # set k = 1
        super().__init__(1, graph, alpha, beta)
        # make sure gamma larger than maximum degree times alpha
        maxDegree = max(self.graph.degree(p) for p in self.graph.nodes)
        self.gamma = maxDegree * alpha + 1

    def checkDomination(self) -> bool:
        """ check whether dominaion condition is met
            it turns out that single domination is equivilent to k-domination with k = 1
        """
        return super().checkDomination()

    def checkIndependence(self) -> bool:
        """check that the dominaotors are independent"""
        return all(self.graph.neighbors(d).isdisjoint(self.dominators) for d in self.dominators)
```

Utils for Part 1.2

```python
def getUtil(self, player: str) -> float:
    def g(i):
        numDominatorIncludeSelf = self.numDominator[i]
        # count self
        numDominatorIncludeSelf += 1 if i in self.dominators else 0
        return self.alpha if numDominatorIncludeSelf == 1 else 0

    if player in self.dominators:
        # gain for dominating neighboring nodes
        ans = sum(g(i) for i in itertools.chain(
            self.graph.neighbors(player), [player]))
        # penalty for being a dominator
        ans -= self.beta
        # panelty for violating independence, (assymetric, only cares neighbors with higher degrees)
        ans -= sum(self.gamma for n in self.graph.neighbors(player)
                   if n in self.dominators and self.graph.degree(n) >= self.graph.degree(player))  #
    else:
        ans = 0
    return ans
```

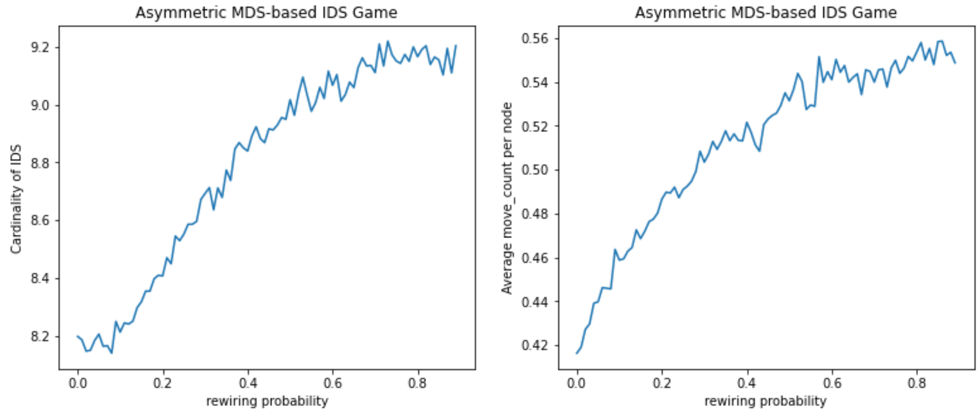
Result for Part 1.2

Average Over 100 Runs, `rewiring_ probability`'s step-size = 0.2

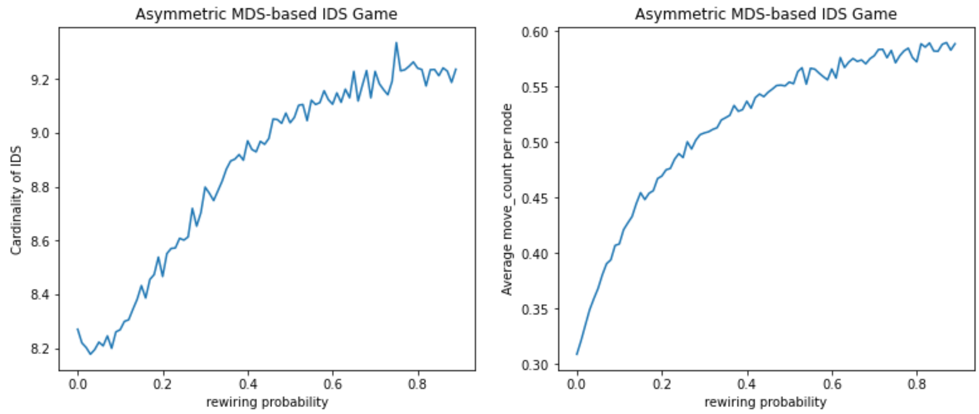


Average 1000 Runs , and `rewiring_ probability`'s step-size = 0.01 (takes about 90 seconds on my laptop)

With Random Initialization



Without Random Initialization



Part 2

We skip the code for class `MaximalMatchingGame` here, please refer to `graph_game.py` for more details.

General

- In the following description of my utility function, the feature is seperated into 3 levels:
  1. Basic
     - (all the formula above excluding `#degree-panelty` and `#robbing-reward`)
  2. Basic + Deg
     - (basic formula + formula with `#degree-panelty`)
  3. Basic + Deg + Rob
     - (all formulat including `#degree-panelty` and `#robbing-reward`)
- The code of the utility is shown here:

```python
def getUtil(self, player: Player) -> float:
    util = 0
    if self.matched(player):
        util += self.alpha
    elif self.strategy[player] and not self.matched(self.strategy[player]):
        util += self.beta
    elif self.strategy[player] == None:
        util += self.delta

    def deg_penalty(n):
        return (self.maxDeg - self.graph.degree(n)) / self.maxDeg

    if self.deg_penalty:
        # match with proposed neigbour with lower degree
        util -= sum(deg_penalty(n) for n in self.graph.neighbors(player) if self.strategy[n] == player and self.strategy[player] != n)
        # proposed to the unmatched neighbor with lower degree
        util -= sum(deg_penalty(n) for n in self.graph.neighbors(player) if not self.matched(n) and self.strategy[player] != n)
        # rob the robbable neighbor with lower degree
        util -= sum(deg_penalty(n) for n in self.graph.neighbors(player) if self.robbable(player, n) and self.strategy[player]!= n)

    if self.robbing_reward and self.deg_penalty:
        """ robbing only work with deg_penalty == True, otherwise, NE may not be a Valid Matching because matched neighbors won't be robbed"""
        if self.robbable(player, self.strategy[player]):
            util += self.gamma
    return util
```

- The code for running experiment is here:

```python
print("== Part 2" + "="*20)
for util_setting in [(False, False), (True, False), (True, True)]:
    print("-"*30)
    print(f'Game = Maximal Matching Game')
    print(
        f'Utils: deg_panelty={util_setting[0]}, robbing_reward={util_setting[1]}')
    print(
        f'{"rewiring_prob":15}, {"move_counts per node":15}, {"matching_counts":15}')
    for rewire_prob_times_10 in range(0, 10, 2):
        rewiring_prob = rewire_prob_times_10 / 10
        move_counts = []
        matching_counts = []
        for i in range(100):
            g = graph.randomWSGraph(
                n=30, k=4, link_rewiring_prob=rewiring_prob)
            gg = MaximalMatchingGame(
                g, deg_penalty=util_setting[0], robbing_reward=util_setting[1])
            gg.randomInit()
            move_count = gg.solve(bestResponseSolver)
            matching_count = gg.numMatchingPairs()
            move_counts.append(move_count)
            matching_counts.append(matching_count)
            assert gg.checkMaximalMatching()
        print(
            f'{rewiring_prob:15.2f}, {sum(move_counts)/100 / 30:15.2f}, {sum(matching_counts)/100:15.2f}')
```

Design Motivation

First I design a strategy for each player (node) to place their action based on the state of their neighbors, the rules are stated as follows:

- Strategy for each node p_i :
  1. If (there exist one neighboring node p_j that is pointing to us , i.e. $c_j = p_i$)
     - choose $c_i = p_j$ to form a pair
     - #degree-panelty choose the neighbor with lower degree to enable more matching
  2. Else if (there exists one neighboring node pj that is not yet paired)
     - choose $c_i = p_j$ to wait for the partner
     - #degree-panelty choose the neighbor with lower degree to enable more matching
  3. Else (all neighboring nodes are paired)
     - #robbing-reward If (there exists one neighbor node p_j that is pairing with a node with higher degree then us, i.e. $c_j = p_k$, $degree(p_k) > degree(p_i)$
       - #degree-panelty #robbing-reward choose $c_i = p_j$ and stole p_j from p_k, because we know p_j will prefer us.
     - Else
       - choose $c_i = null$

Utility Definition

- Utilitiy for $u_i(c_i)$, and parameters $\alpha > \beta + 1, \beta > \gamma + 1, \gamma > \delta + 1, \delta > 1$
  1. If $(c_j = p_i) \wedge (c_i = p_j)$ for some j we receive matching reward

$$+\alpha > \beta + 1$$

     - #degree-panelty For each k with $(c_k = p_i) \wedge (c_i \neq p_k)$ we receive penalty

$$-1 \leq \frac{-(\max_{p \in P}[degree(p)] - degree(c_k))}{\max_{p \in P}[degree(p)]} \leq 0$$

     So that higher the degree of the abandoned c_k , lower the penalty
  2. If $c_i = p_j$, and p_j is not matched for some j, we receive waiting reward

$$+\beta > \gamma + 1$$

   . #degree-panelty For each k with (p_k not matched) $\wedge (p_k \in N_i) \wedge (c_i \neq p_k)$ we receive penalty

$$-1 \leq \frac{-(\max_{p \in P}[degree(p)] - degree(c_k))}{\max_{p \in P}[degree(p)]} \leq 0$$

3. #robbing-reward If p_j is "robbable" (i.e. $(c_j = p_k) \wedge (c_k = p_j) \wedge (degree(p_k) > degree(p_i))$) and $(c_i = p_j)$ we receive robbing reward

$$+\gamma > \delta + 1$$

   - #degree-panelty #robbing-reward For each "robbable" p_j with $c_i \neq p_j$, we receive penalty

$$-1 \leq \frac{-(\max_{p \in P}[degree(p)] - degree(c_j))}{\max_{p \in P}[degree(p)]} \leq 0$$

   - If $c_i = null$, we receive giving up bonus

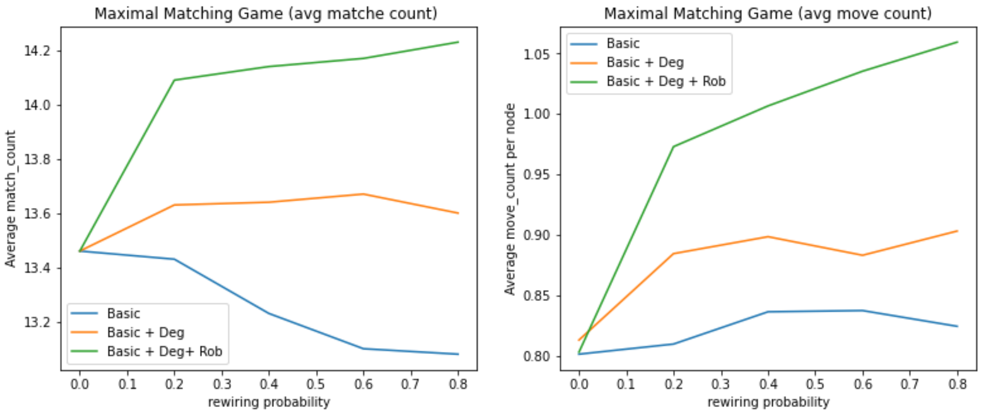
$$+\delta > 1$$

Results

- We show the ablation of different different levels of utility designs:
  1. Basic
     - (all the formula above excluding #degree-panelty and #robbing-reward)
  2. Basic + Deg
     - (basic formula + formula with #degree-panelty)
  3. Basic + Deg + Rob
     - (all formulat including #degree-panelty and #robbing-reward)

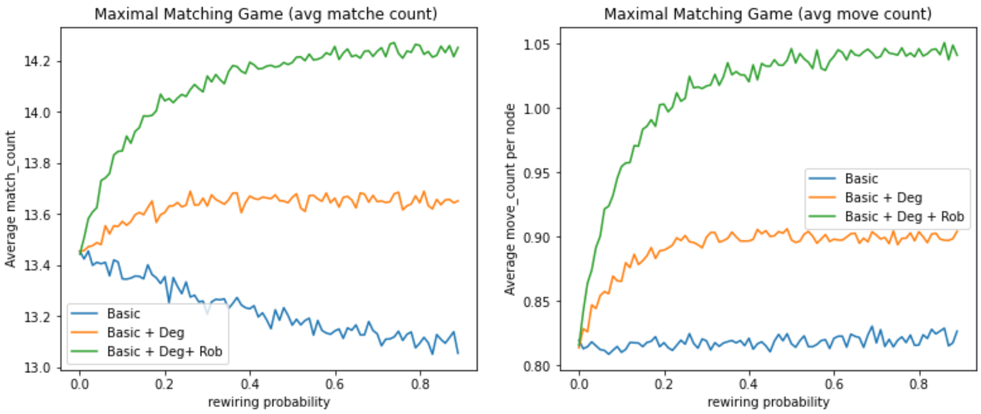
Average over 100 runs, `rewiring_ probability`'s step-size = 0.1

With Random Initialization

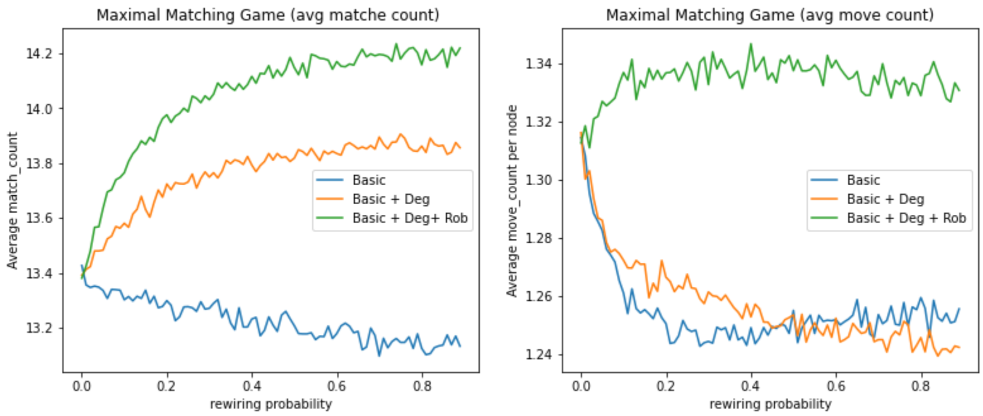


Average over 1000 Runs, `rewiring_ probability`'s step-size = 0.01 (takes about 15 mins on my laptop)

With Random Initialization



Without Random Initialization



We can see that the full version runs the slowest, but returns the best matching solution
Random initialization helps the model converge faster.