Game Theory and Its Application

Homework1

My problems are Multi-Domination Game, Symmetric MDS-based IDS game and Maximal Matching.

Student ID: 311551140 Name: 陳品竣

The following report mainly introduces the content of sample code, the result of simulation and my interpretation in the Multi-Domination Game, Symmetric MDS-based IDS game and Maximal Matching.

- Simulating the execution of graph games based on your student ID

Student ID mod 6	Game to simulate
0	Multi-Domination Game
1	k-Domination Game
2	Maximal Independent Set (MIS) Game (Symmetric)
3	Asymmetric MIS Game
4	Weighted MIS Game
5	MIS-based IDS Game

Student ID mod 2	Game to simulate
0	Symmetric MDS-based IDS Game
1	Asymmetric MDS-based IDS Game

In the beginning, the WS model with a 30-node regular graph is first formed. After that, each node has 4 edges connecting to its 4 nearest neighbors. The sample code to build the WS model in next page.

```
    for(int i = 0; i < n; i ++)
    {
        if(i > 1 && i < 28)
        {
            WS[i][i-1] = 1;
            WS[i][i-2] = 1;
            WS[i][i+1] = 1;
            WS[i][i+2] = 1;
        }
        else
        {
            if(i == 0)
            {
                WS[0][28] = 1;
                WS[0][29] = 1;
                WS[0][1] = 1;
                WS[0][2] = 1;
            }
            if(i == 1)
            {
                WS[1][0] = 1;
                WS[1][29] = 1;
                WS[1][2] = 1;
                WS[1][3] = 1;
            }
            if(i == 28)
            {
                WS[28][27] = 1;
                WS[28][26] = 1;
                WS[28][29] = 1;
                WS[28][0] = 1;
            }
            if(i == 29)
            {
                WS[29][27] = 1;
                WS[29][28] = 1;
                WS[29][0] = 1;
                WS[29][1] = 1;
            }
        }
    }
```

After modeling, we have to decide whether the model remains the same connection between nodes or not. Hence, give the probability(*pr*) from 0 to 0.8 step by 0.2. If I random select the edge which in precious state, I won't random choose again. So my edge could decrease. It determines whether to rewire the link between nodes in the previous state or not. The sample code is as below.

```
void rewire_function(int WS[30][30], int rewired[30][30], double &p)
{
    for(int i = 0; i < 30; i++)
        for(int j = 0; j < 30; j++)
            rewired[i][j] = 0;
    double rand_p;
    for(int i = 0; i < 30; i++)
    {
        for(int j = i + 1; j < 30; j++)
        {
            // if have edge then rewired
            if(WS[i][j] == 1)
            {
                rand_p = (double)rand() / RAND_MAX;
                if(rand_p <= p)
                {
                    int a = 0;
                    int b = 0;
                    while(a == b)
                    {
                        a = rand() % 30;
                        b = rand() % 30;
                    }
                    rewired[a][b] = 1;
                    rewired[b][a] = 1;
                }
                else
                {
                rewired[i][j] = 1;
                rewired[j][i] = 1;
                }
            }
        }
    }
```

Notice that since the graph of WS model is undirected (i.e. all of its edges are bi-directional), the adjacency matrix is a (0, 1)-matrix with zeros on its diagonal and symmetric. And then, we initialize the game state randomly with uniform distribution. If the node is in our strategy, then we set the value with 1; otherwise, we set the value with 0.

```
int player_strategy[30] = {0}; // out = 0 , in = 1
```

Next, I will first explain the ***Multi-Domination Game*** of requirement 1-1. First, I randomize the k[j] of each node which k is [1,3] and the game state. The sample code is as below.

```
for(int i = 0; i < 30; i++)
{
    // kj be a random integer within the range [1,3]
    k[i] = rand() % 3 + 1;


    // randomize initial game state, out = 0, in = 1
    player_strategy[i] = rand() % 2;
}
```

Then I use a while loop to determine whether the Nash equilibrium is reached. Randomly select a player (this player cannot be selected in the loop unless a player has changed their decision). The sample code is as below.

```
while(NE != true)
{
    // random pick up one player who can improve its utility
    pi = rand() % 30;
    while(player_state[pi] == 1)
    {
        pi = rand() % 30;
    }
    player_state[pi] = 1;
```

Then calculate the numbers of the selected node which dominated by its closed neighbors. The sample code is as below.

```
// count now pi have which dominations int his closed neighbors
    int nums_domination = 0;
    for(int i = 0; i < 30; i++)
    {
        if(pi == i)
        {
            if(player_strategy[pi] == 1)
                nums_domination++;
            continue;
        }
        if(rewired[pi][i] == 1)
        {
            if(player_strategy[i] == 1)
                nums_domination++;
        }
    }
```

If pi in the set at original and dominations bigger than k[pi], then pi out the set. If pi out the set at original and dominations smaller than k[pi], then pi in the set. If pi changes its strategy, then move count plus one and all players need to determine their strategy again. The sample code is as below.

```
// pi in the set
if (player_strategy[pi] == 1)
{
    // dominations > k[pi] => pi is redundant
    if(nums_domination > k[pi])
    {
        player_strategy[pi] = 0;
        move_count++;

        //because pi change the strategy so all player decide in or out
        for(int i = 0; i < 30; i++)
            player_state[i] = 0;
        player_state[pi] = 1;
        continue;
    }
}

// pi out the set
else
{
    if(nums_domination < k[pi])
    {
        player_strategy[pi] = 1;
        move_count++;
        //because pi change the strategy so all player decide in or out
        for(int i = 0; i < 30; i++)
            player_state[i] = 0;
        player_state[pi] = 1;
        continue;
    }
}
```

Finally, I test the all player whether their strategies and never change their strategies, so the game reach Nash Equilibrium. The sample code is as below.

```
    // test the all player never change their strategy
    int flag = 0;
    for(int i = 0; i < 30; i++)
    {
        if(player_state[i] == 0)
            flag = 1;
    }

    if(flag == 0)
        NE = true;
}
```

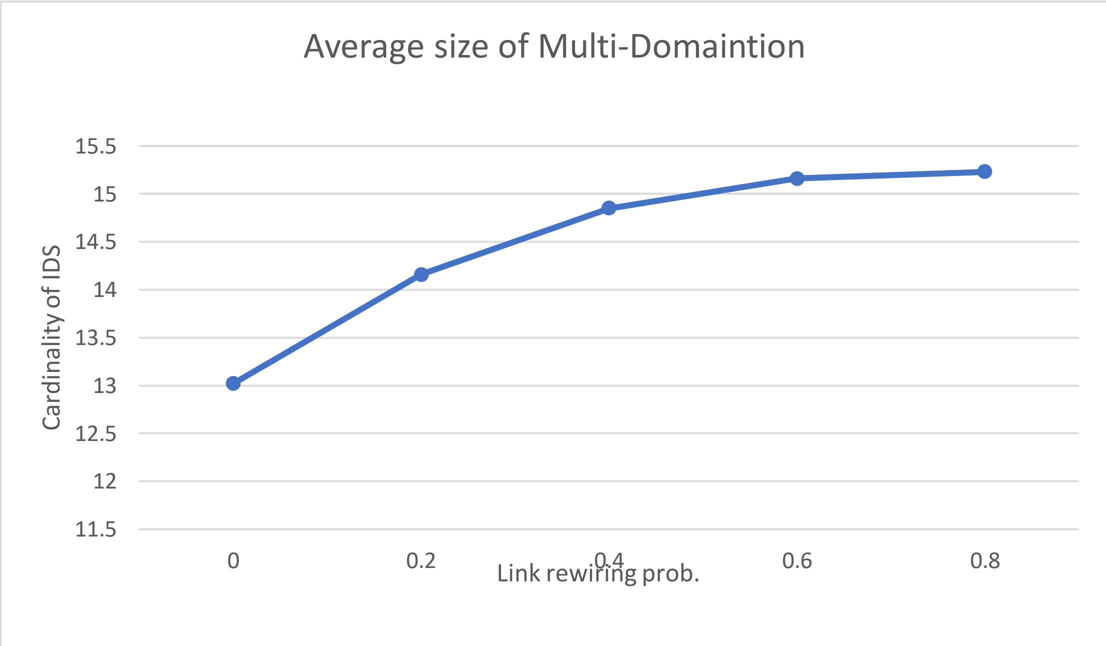
And calculate the final game state's cardinality. The sample code is as below.

```
    for(int i = 0; i < 30; i++)
    {
        if(player_strategy[i] == 1)
            cardinality++;
    }
```

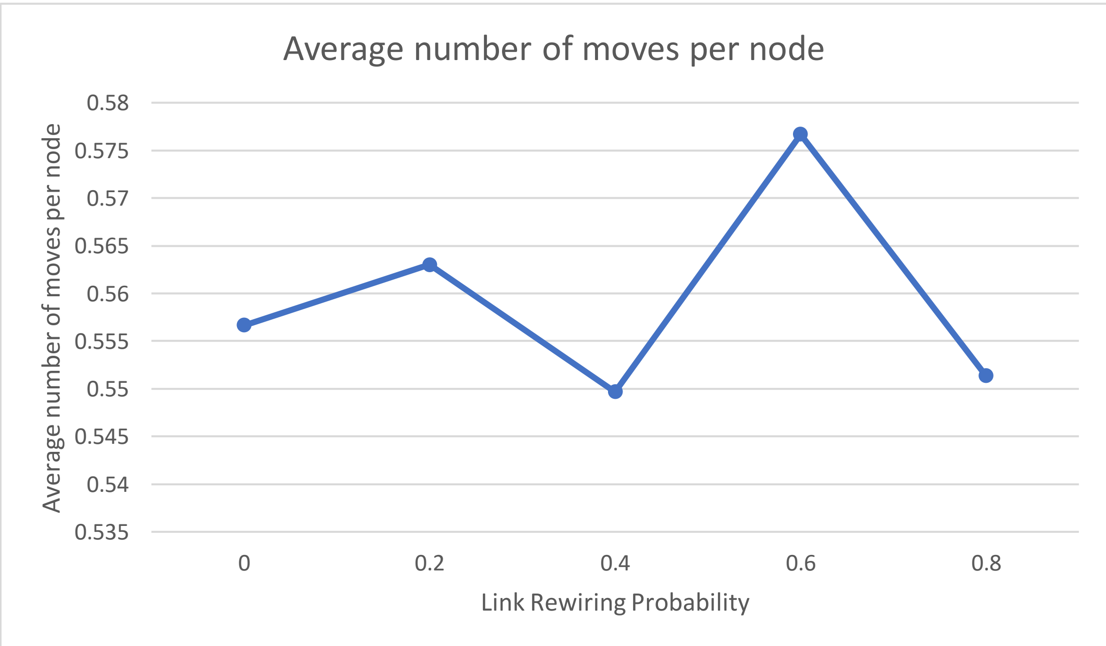
I have a check function to check when reach to Nash Equilibrium, the game state need to satisfy the Multi-Domination game definition. But the default random k[i]=[1,3], and the graph which rewired could has isolated node or the component which has only two nodes. If their k are 2 or 3, they can't satisfy the Multi-Domination game definition. This check function almost never satisfy! The sample code is as below.

```
bool check_Multi_Domination(int rewired[30][30], int player_strategy[30], int k[30])
{
    for(int i = 0; i < 30; i++)
    {
        int domination = 0;

        // pi in the set
        if(player_strategy[i] == 1)
        {
            domination++;
            for(int j = 0; j < 30; j++)
            {
                if(rewired[i][j] == 1)
                {
                    if(player_strategy[j] == 1)
                        domination++;
                }
            }
            if(domination < k[i])
                return false;
        }

        // pi out the set
        else
        {
            for(int j = 0; j < 30; j++)
            {
                if(rewired[i][j] == 1)
                {
                    if(player_strategy[j] == 1)
                        domination++;
                }
            }
            if(domination < k[i])
                return false;
        }
    }
    return true;
}
```

In the process of execution, we take the utility of this node to decide whether to change decision or not. Moreover, we count the movement when our strategy profile changes. After execution 100 times for each rewire probability (*pr*), we can get the graph as follows.



Looking at the line chart above, we will find that as the probability value increases, the cardinality of IDS will also increase. It indicates that as long as rewiring, the cardinality of IDS may be influenced. Because my rewired method could result to reduce the edge number so the rewired graph maybe has many components. If the graph has many component then it need to more cardinality to domination the graph.



Next, this line chart shows that the probability value can't affect the move count. The error in the tolerance scope. Although the WS model pass through rewired, we also have 30 nodes. Reach the Nash Equilibrium only need about 0.55 move count per node.

Next, I will explain the ***Symmetric MDS-based IDS game*** of requirement 1-2. I also use a while loop to determine whether the Nash equilibrium is reached. Randomly select a player (this player cannot be selected in the loop unless a player has changed their decision). The sample code is as below.

```
while(NE != true)
{
    // random pick up one player who can improve its utility
    pi = rand() % 30;
    while(player_state[pi] == 1)
    {
        pi = rand() % 30;
    }
    player_state[pi] = 1;

```

If pi out of set, I check the set of its neighbor. If its neighbor has one or more node in the set then pi also select out of set. Otherwise pi join to the set. The sample code is as below.

```
    // if pi out of set
    if(player_strategy[pi] == 0)
    {
        int neighbor_in_set = 0;
        for(int i = 0; i < 30; i++)
        {
            // check the neighbor of pi
            if(rewired[pi][i] == 1)
            {
                // if the neighbor of pi in the set then pi need not to join the set
                if(player_strategy[i] == 1)
                {
                    neighbor_in_set = 1;
                    break;
                }
            }
        }

        // if all neighbor of pi are not in the set then pi join the set
        if(neighbor_in_set == 0)
        {
            player_strategy[pi] = 1;
            move_count++;
            //because pi change the strategy so all player decide in or out
            for(int i = 0; i < 30; i++)
                player_state[i] = 0;
            player_state[pi] = 1;
            continue;
        }
    }
```

If pi in the set, I check the set of its neighbor. If its neighbor has one or more node in the set then pi select out of set. Otherwise pi also in the set. The sample code is as below.

```
// if pi in the set -> player_strategy[pi] = 1
else
{
    int neighbor_in_set = 0;
    for(int i = 0; i < 30; i++)
    {
        // check the neighbor of pi
        if(rewired[pi][i] == 1)
        {
            // so pi need not to join the set
            if(player_strategy[i] == 1)
            {
                neighbor_in_set = 1;
                break;
            }
        }
    }
    if(neighbor_in_set == 1)
    {
        player_strategy[pi] = 0;
        move_count++;
        //because pi change the strategy so all player decide in or out
        for(int i = 0; i < 30; i++)
            player_state[i] = 0;
        player_state[pi] = 1;
        continue;
    }
}
```

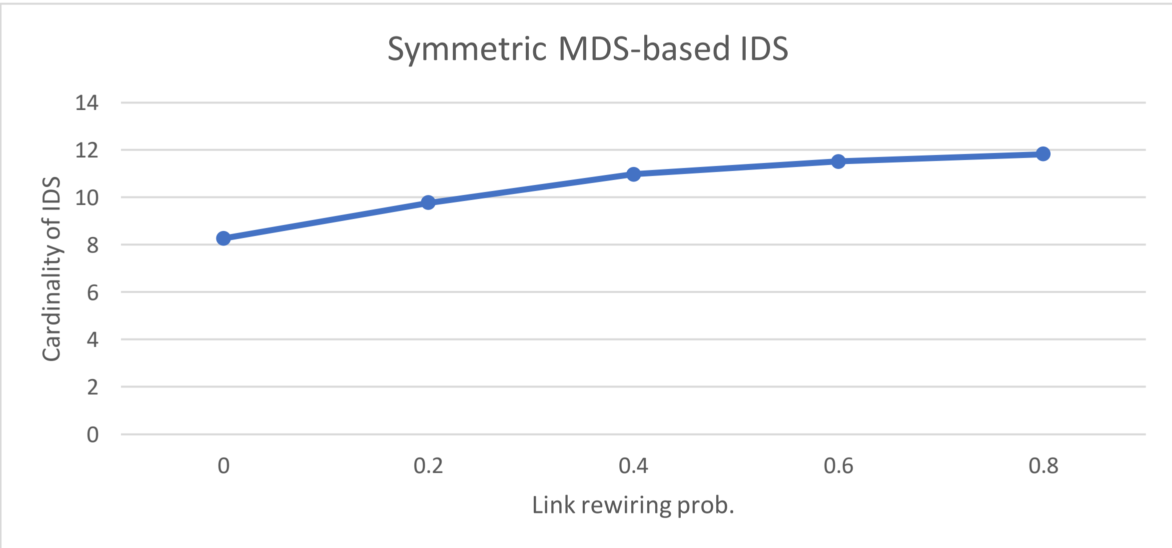
Finally, I test the all player whether their strategies and never change their strategies, so the game reach Nash Equilibrium. The sample code is as below.

```
    // test the all player never change their strategy
    int test_NE = 0;
    for(int i = 0; i < 30; i++)
    {
        if(player_state[i] == 0)
            test_NE = 1;
    }
    if(test_NE == 0)
        NE = true;
}
```

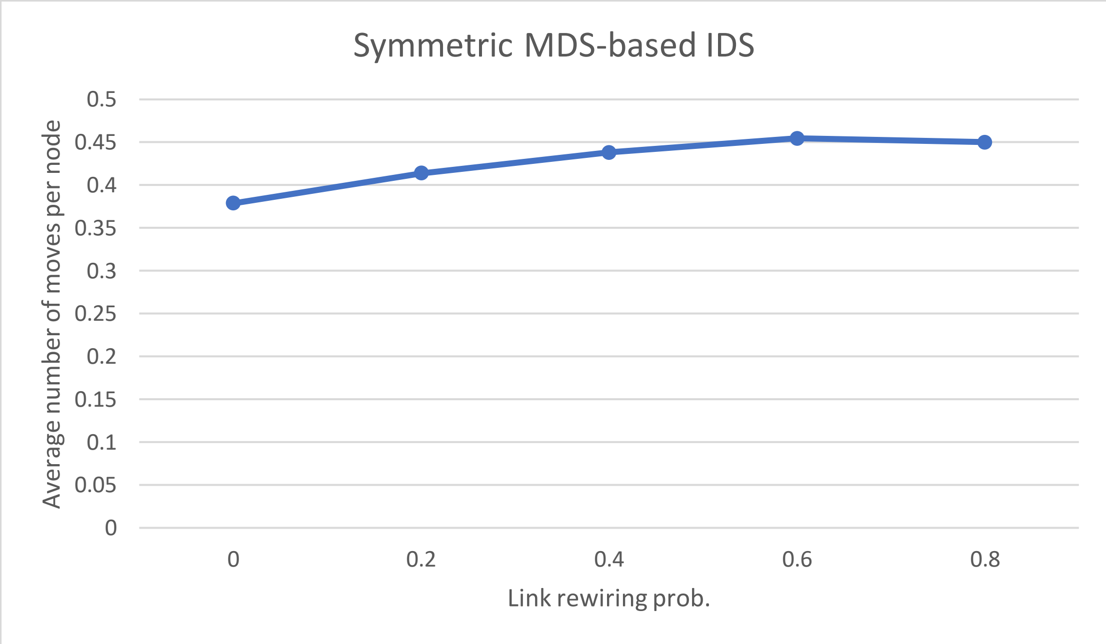
I have a check function to check when reach to Nash Equilibrium, the game state need to satisfy the Symmetric MDS-based IDS game definition. The sample code is as below.

```
bool check_Symmetric_MDS_based_IDS(int rewired[30][30], int player_strategy[30])
{
    for(int i = 0; i < 30; i++)
    {
        // pi in the set
        if(player_strategy[i] == 1)
        {
            for(int j = 0; j < 30; j++)
            {
                if(rewired[i][j] == 1)
                {
                    if(player_strategy[j] == 1)
                        return false;
                }
            }
        }

        // pi out the set
        else
        {
            int flag = 0;
            for(int j = 0; j < 30; j++)
            {
                if(rewired[i][j] == 1)
                {
                    if(player_strategy[j] == 1)
                        flag = 1;
                }
            }
            if(flag == 0)
                return false;
        }
    }
    return true;
}
```

In the process of execution, we take the utility of this node to decide whether to change decision or not. Moreover, we count the movement when our strategy profile changes. After execution 100 times for each rewire probability (*pr*), we can get the graph as follows.



Looking at the line chart above, we will find that as the probability value increases, the cardinality of IDS will also increase. It indicates that as long as rewiring, the cardinality of IDS may be influenced. Because my rewired method could result to reduce the edge number so the rewired graph maybe has many components. If the graph has many component then it need to more cardinality to domination the graph.



Next, this line chart shows that the probability value can't affect the move count. The error in the tolerance scope. Although the WS model pass through rewired, we also have 30 nodes. Reach the Nash Equilibrium only need about 0.37 to 0.45 move count per node.

Next, I will explain the ***Maximal Matching*** of requirement 2. Before starting the game, we have to design the utility function for each player as follows.

$$u_i(C) = \begin{cases} 0 & ,\text{if } c_i \neq -1 \vee \forall p_j \in N_i,\ c_j \neq -1 \\ n - \sum\limits_{p_j \in N_i} g_j(C) & ,\text{otherwise} \end{cases}$$

$$\text{where } g_j(C) = \begin{cases} 0 & ,\text{if } c_j \neq -1 \\ 1 & ,\text{otherwise} \end{cases}$$

In order to increase the number of matched pairs, I give high priority to nodes with few neighbors in matching. The function is like a penalty for the purpose of decreasing the priority to nodes with more neighbors in matching. If the node is not in pairs and its open neighbors have few neighbors in matching, then it would have high utility value to match. After the utility function, the following code is to check whether the game state achieves the Nash equilibria or not.

```
    //record the matching of node, init the node not in the matching so they are -1
    int matching[30];
    for(int i = 0; i < 30; i++)
        matching[i] = -1;

    // randomize the inital state, the state also a matching
    int left, right;
    while(edge > 0)
    {
        left = rand() % 30;
        right = rand() % 30;

        //random select two node which need to connect
        while(left == right || rewired[left][right] == 0)
        {
            left = rand() % 30;
            right = rand() % 30;
        }

        // two node need to not in the matching and rand need to = 1   -> then the edge can join to the init matching
        if(matching[left] != -1 || matching[right] != -1 || rand() % 2 == 0)
        {
            edge--;
            continue;
        }

        // the edge join to the init matching
        matching[left] = right;
        matching[right] = left;
        edge--;
    }
```

After introducing the WS model, utility function, we can take all of these to construct the main part. The main content of the code is as follows.

```
    while(!NE)
    {
        // random pick up one player who can improve its utility
        pi = rand() % 30;
        while (player_state[pi] == 1)
        {
            pi = rand() % 30;
        }
        player_state[pi] = 1;

        // pi in the matching but pi can choose the lower degree neighbor which not in the matching and pair it
        if(matching[pi] > -1)
        {
            int min_degree = degree[matching[pi]];
            int select_node = matching[pi];  //original pair
            bool choose = false;
            for(int i = 0; i < 30; i++)
            {
                if(rewired[pi][i] == 1)
                {
                    if(degree[i] < min_degree && matching[i] == -1)
                    {
                        min_degree = degree[i];
                        select_node = i;
                        choose = true;
                    }
                }
            }
            matching[pi] = select_node;
            matching[select_node] = pi;
            if(choose)
            {
                move_count++;
                for (int i = 0; i < 30; i++)
                    player_state[i] = 0;
                player_state[pi] = 1;
            }
        }
```

```
//matching[pi] = -1 , pi not in the matching so pi can choose the min degree node which not in the matching and pair it
else
{
    int min_degree = 100;
    int select_node;
    bool choose = false;
    for(int i = 0; i < 30; i++)
    {
        if(rewired[pi][i] == 1)
        {
            if(degree[i] < min_degree && matching[i] == -1)
            {
                min_degree = degree[i];
                select_node = i;
                choose = true;
            }
        }
    }
    if(choose)
    {
        matching[pi] = select_node;
        matching[select_node] = pi;
        move_count++;
        for (int i = 0; i < 30; i++)
            player_state[i] = 0;
        player_state[pi] = 1;
    }
}
```

Finally, I test the all player whether their strategies and never change their strategies, so the game reach Nash Equilibrium. The sample code is as below.

```
// test the all player never change their strategy
int flag = 0;
for (int i = 0; i < 30; i++)
{
    if (player_state[i] == 0)
        flag = 1;
}

if (flag == 0)
    NE = true;
```

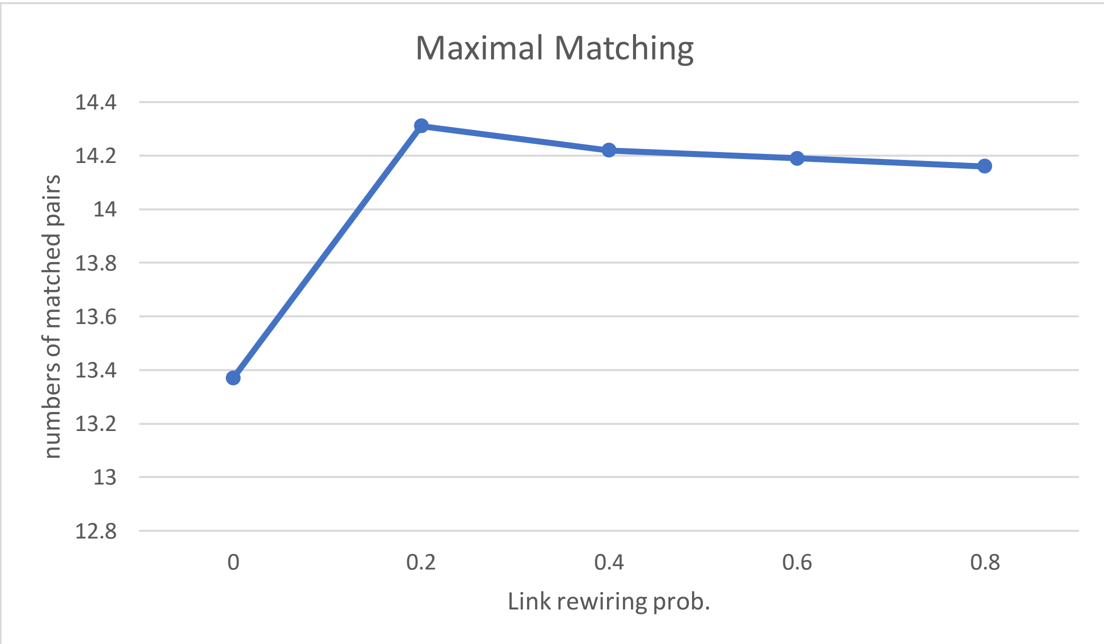
And I calculate the numbers of pair.

```
for(int i = 0; i < 30; i++)
    {
        if(matching[i] != -1)
            pair++;
    }
    pair = pair / 2;
```

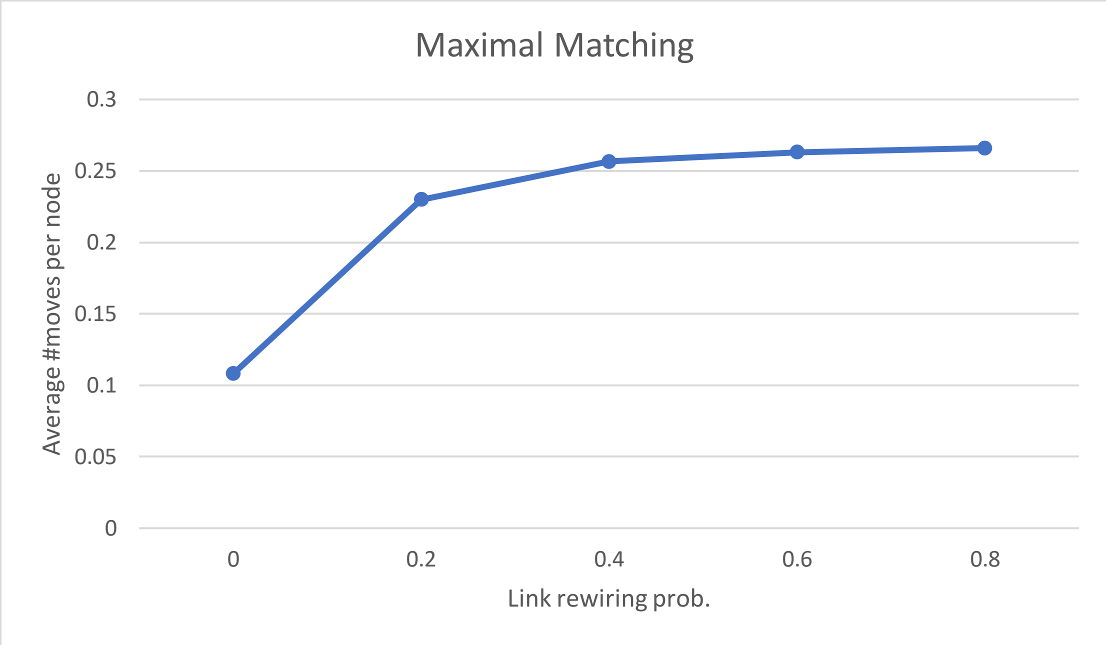
I have a check function to check when reach to Nash Equilibrium, the game state need to satisfy the Maximal Matching game definition. The sample code is as below.

```
bool check_Maximal_Matching(int rewired[30][30], int matching[30])
{
    for(int i = 0; i < 30; i++)
    {
        for(int j = 0; j < 30; j++)
        {
            if(rewired[i][j] == 1)
            {
                if(matching[i] == j && matching[j] == i)
                {
                    for(int k = 0; k < 30; k++)
                    {
                        if(k == i || k == j)
                            continue;
                        else
                            if(matching[k] == i && matching[k] == j)
                                return false;
                    }
                }
            }
        }
    }
    return true;
}
```

In the process of execution, we take the utility of this node to decide whether to change decision or not. Moreover, we count the movement when our strategy profile changes. After execution 100 times for each rewire probability (*pr*), we can get the graph as follows.



Looking at the line chart above, we will find that the pairs increases at beginning, but As the probability of rewired increases, the pairs decrease. Because my rewired function could result to reduce the edge numbers, so pairs also decrease. It indicates that as long as rewiring, the number of matched pairs may be influenced. Hence, it shows the specific relationship between both of them.



Next, the above line chart shows that as the probability value increases, the average number of moves per node will increase as well. It indicates that as long as rewiring, it would take more steps to find out the Nash equilibria as our strategy profile. Therefore, it shows the specific relationship between both of them.