

# 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.

```

542  ✓   for(int i = 0; i < n; i ++)  

543      {  

544  ✓       if(i > 1 && i < 28)  

545           {  

546               WS[i][i-1] = 1;  

547               WS[i][i-2] = 1;  

548               WS[i][i+1] = 1;  

549               WS[i][i+2] = 1;  

550           }  

551  ✓       else  

552           {  

553  ✓           if(i == 0)  

554               {  

555                   WS[0][28] = 1;  

556                   WS[0][29] = 1;  

557                   WS[0][1] = 1;  

558                   WS[0][2] = 1;  

559               }  

560  ✓           if(i == 1)  

561               {  

562                   WS[1][0] = 1;  

563                   WS[1][29] = 1;  

564                   WS[1][2] = 1;  

565                   WS[1][3] = 1;  

566               }  

567  ✓           if(i == 28)  

568               {  

569                   WS[28][27] = 1;  

570                   WS[28][26] = 1;  

571                   WS[28][29] = 1;  

572                   WS[28][0] = 1;  

573               }  

574  ✓           if(i == 29)  

575               {  

576                   WS[29][27] = 1;  

577                   WS[29][28] = 1;  

578                   WS[29][0] = 1;  

579                   WS[29][1] = 1;  

580               }  

581           }  

582      }

```

After modeling, we have to decide whether the model remains the same connection between nodes or not. Hence, give the probability( $p_r$ ) 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 rewired the link between nodes in the previous state or not. The sample code is as below.

```
10 void rewired_function(int WS[30][30], int rewired[30][30], double &p)
11 {
12     for(int i = 0; i < 30; i++)
13     {
14         for(int j = 0; j < 30; j++)
15             rewired[i][j] = 0;
16     }
17     double rand_p;
18     for(int i = 0; i < 30; i++)
19     {
20         for(int j = i + 1; j < 30; j++)
21         {
22             // if have edge then rewired
23             if(WS[i][j] == 1)
24             {
25                 rand_p = (double)rand() / RAND_MAX;
26                 if(rand_p <= p)
27                 {
28                     int a = 0;
29                     int b = 0;
30                     while(a == b)
31                     {
32                         a = rand() % 30;
33                         b = rand() % 30;
34                     }
35                     rewired[a][b] = 1;
36                     rewired[b][a] = 1;
37                 }
38             }
39             else
40             {
41                 rewired[i][j] = 1;
42                 rewired[j][i] = 1;
43             }
44         }
45     }
46 }
```

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.**

```
105      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.

```
109      for(int i = 0; i < 30; i++)
110      {
111          // kj be a random integer within the range [1,3]
112          k[i] = rand() % 3 + 1;
113
114
115          // randomize initial game state, out = 0, in = 1
116          player_strategy[i] = rand() % 2;
117      }
```

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.

```
119      while(NE != true)
120      {
121          // random pick up one player who can improve its utility
122          pi = rand() % 30;
123          while(player_state[pi] == 1)
124          {
125              pi = rand() % 30;
126          }
127          player_state[pi] = 1;
```

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

```

128 // count now pi have which dominations int his closed neighbors
129 int nums_domination = 0;
130 for(int i = 0; i < 30; i++)
131 {
132     if(pi == i)
133     {
134         if(player_strategy[pi] == 1)
135         |     nums_domination++;
136         continue;
137     }
138     if(rewired[pi][i] == 1)
139     {
140         if(player_strategy[i] == 1)
141         |     nums_domination++;
142     }
143 }

```

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.

```

145 // pi in the set
146 if (player_strategy[pi] == 1)
147 {
148     // dominations > k[pi] => pi is redundant
149     if(nums_domination > k[pi])
150     {
151         player_strategy[pi] = 0;
152         move_count++;
153
154         //because pi change the strategy so all player decide in or out
155         for(int i = 0; i < 30; i++)
156         |     player_state[i] = 0;
157         player_state[pi] = 1;
158         continue;
159     }
160 }
161
162 // pi out the set
163 else
164 {
165     if(nums_domination < k[pi])
166     {
167         player_strategy[pi] = 1;
168         move_count++;
169         //because pi change the strategy so all player decide in or out
170         for(int i = 0; i < 30; i++)
171         |     player_state[i] = 0;
172         player_state[pi] = 1;
173         continue;
174     }
175 }

```

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.

```

178      // test the all player never change their strategy
179      int flag = 0;
180      for(int i = 0; i < 30; i++)
181      {
182          if(player_state[i] == 0)
183          |   flag = 1;
184      }
185
186      if(flag == 0)
187      |   NE = true;
188  }

```

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

```

189      for(int i = 0; i < 30; i++)
190      {
191          if(player_strategy[i] == 1)
192          |   cardinality++;
193      }

```

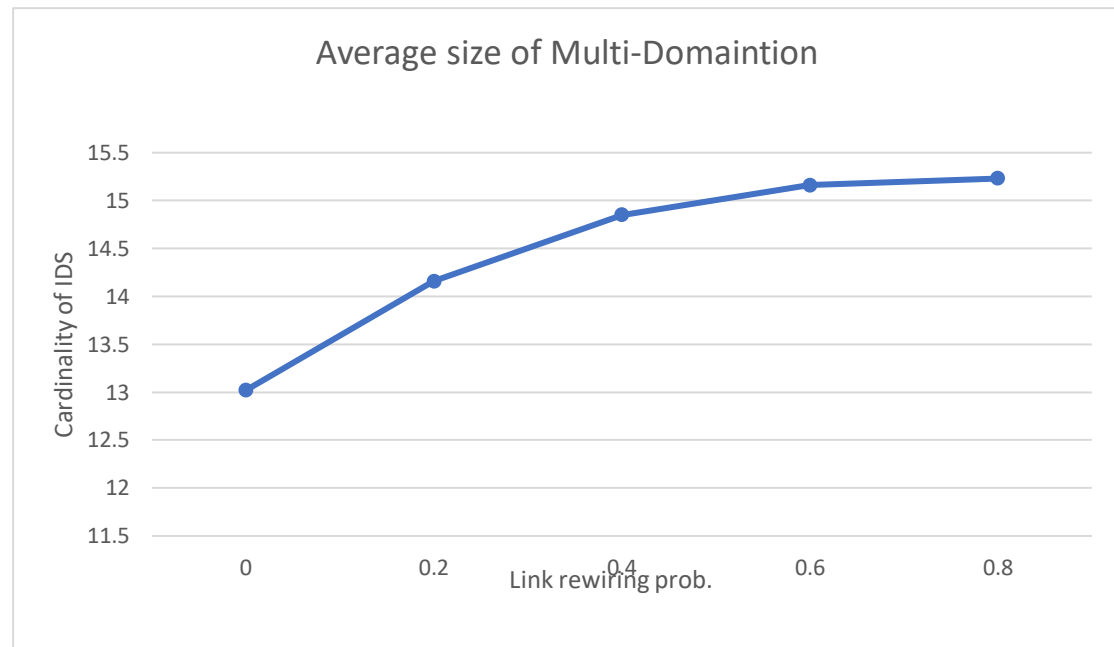
I have a check function to check when reach to Nash Equilibrium, the game state need to satisfy the Multi-Domination game definition. But the default random  $k[i]=[1,3]$ , and the graph which rewired could has isolated node or the component which has only two nodes. If their  $k$  are 2 or 3, they can't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

```

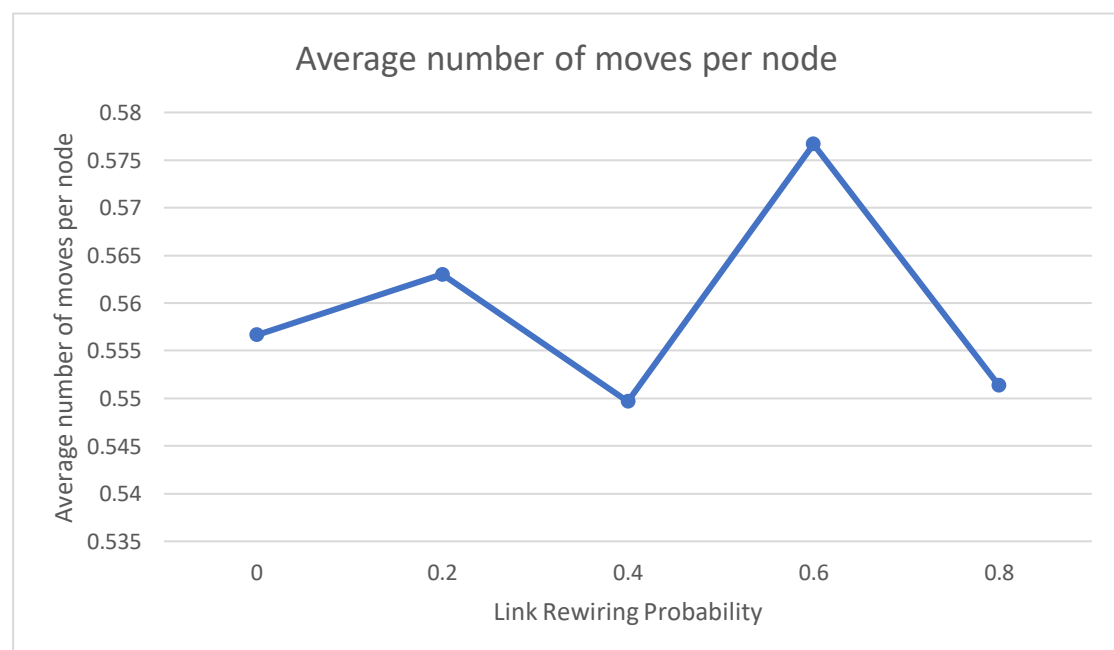
57  bool check_Multi_Domination(int rewired[30][30], int player_strategy[30], int k[30])
58  {
59      for(int i = 0; i < 30; i++)
60      {
61          int domination = 0;
62
63          // pi in the set
64          if(player_strategy[i] == 1)
65          {
66              domination++;
67              for(int j = 0; j < 30; j++)
68              {
69                  if(rewired[i][j] == 1)
70                  {
71                      if(player_strategy[j] == 1)
72                      |   domination++;
73                  }
74              }
75              if(domination < k[i])
76              |   return false;
77          }
78
79          // pi out the set
80          else
81          {
82              for(int j = 0; j < 30; j++)
83              {
84                  if(rewired[i][j] == 1)
85                  {
86                      if(player_strategy[j] == 1)
87                      |   domination++;
88                  }
89              }
90              if(domination < k[i])
91              |   return false;
92          }
93      }
94      return true;
95  }

```

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.

```
256 while(NE != true)
257 {
258     // random pick up one player who can improve its utility
259     pi = rand() % 30;
260     while(player_state[pi] == 1)
261     {
262         pi = rand() % 30;
263     }
264     player_state[pi] = 1;
265 }
```

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.

```
266 // if pi out of set
267 if(player_strategy[pi] == 0)
268 {
269     int neighbor_in_set = 0;
270     for(int i = 0; i < 30; i++)
271     {
272         // check the neighbor of pi
273         if(rewired[pi][i] == 1)
274         {
275             // if the neighbor of pi in the set then pi need not to join the set
276             if(player_strategy[i] == 1)
277             {
278                 neighbor_in_set = 1;
279                 break;
280             }
281         }
282     }
283
284     // if all neighbor of pi are not in the set then pi join the set
285     if(neighbor_in_set == 0)
286     {
287         player_strategy[pi] = 1;
288         move_count++;
289         //because pi change the strategy so all player decide in or out
290         for(int i = 0; i < 30; i++)
291         | player_state[i] = 0;
292         player_state[pi] = 1;
293         continue;
294     }
295 }
```



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.

```

297 // if pi in the set -> player_strategy[pi] = 1
298 else
299 {
300     int neighbor_in_set = 0;
301     for(int i = 0; i < 30; i++)
302     {
303         // check the neighbor of pi
304         if(rewired[pi][i] == 1)
305         {
306             // so pi need not to join the set
307             if(player_strategy[i] == 1)
308             {
309                 neighbor_in_set = 1;
310                 break;
311             }
312         }
313     }
314     if(neighbor_in_set == 1)
315     {
316         player_strategy[pi] = 0;
317         move_count++;
318         //because pi change the strategy so all player decide in or out
319         for(int i = 0; i < 30; i++)
320         |     player_state[i] = 0;
321         player_state[pi] = 1;
322         continue;
323     }
324 }

```

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.

```

326 // test the all player never change their strategy
327 int test_NE = 0;
328 for(int i = 0; i < 30; i++)
329 {
330     if(player_state[i] == 0)
331     |     test_NE = 1;
332 }
333 if(test_NE == 0)
334 |     NE = true;
335 }

```

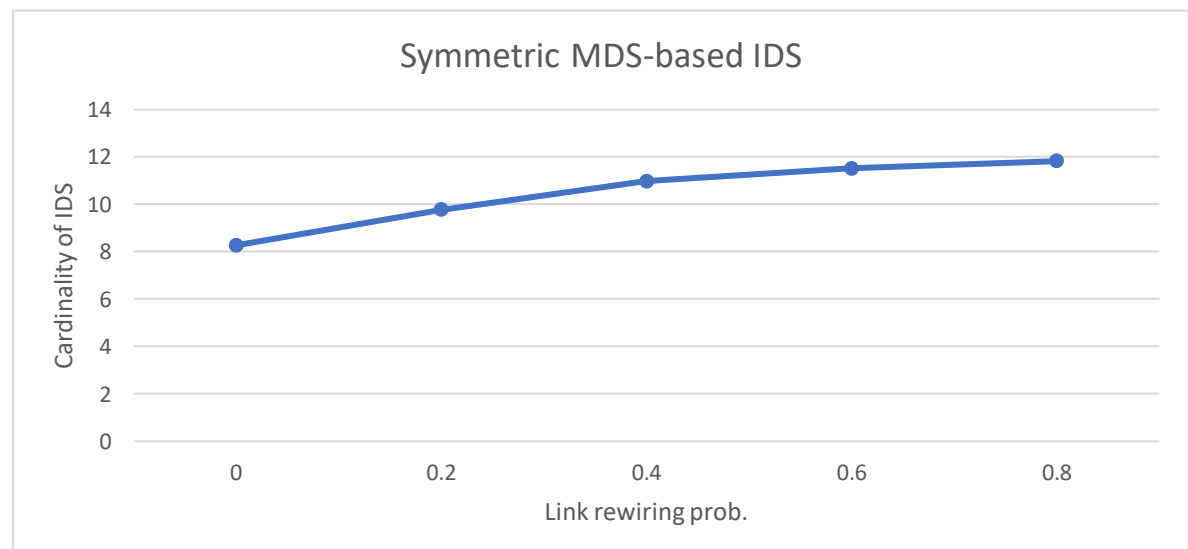
I have a check function to check when reach to Nash Equilibrium, the game state need to satisfy the Symmetric MDS-based IDS game definition. The sample code is as below.

```

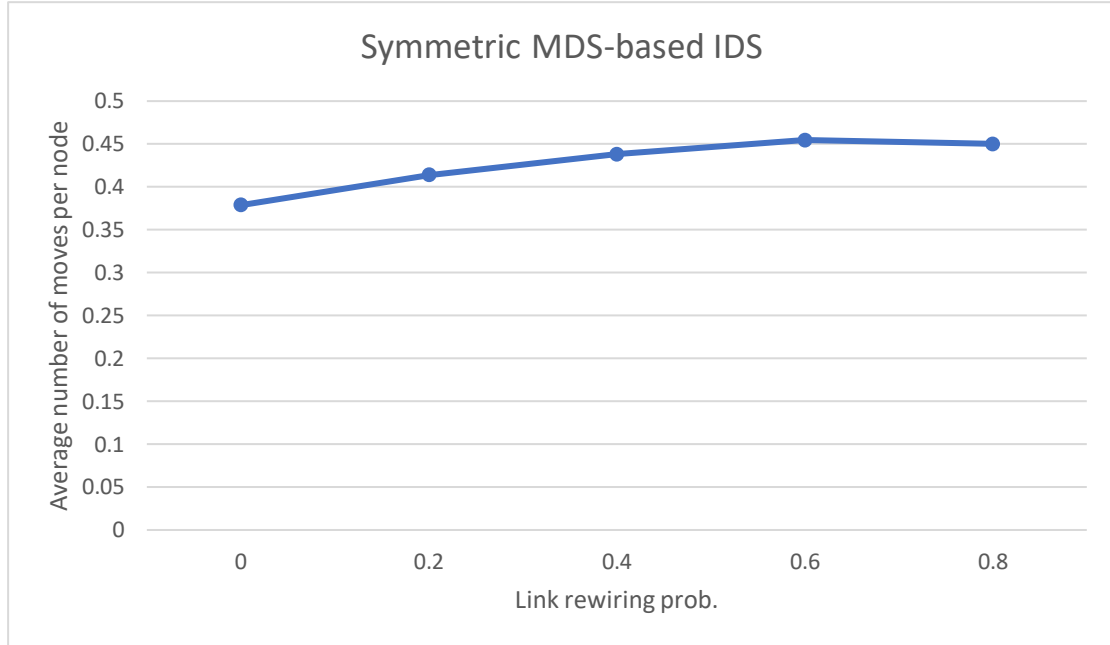
204 bool check_Symmetric_MDS_based_IDS(int rewired[30][30], int player_strategy[30])
205 {
206     for(int i = 0; i < 30; i++)
207     {
208         // pi in the set
209         if(player_strategy[i] == 1)
210         {
211             for(int j = 0; j < 30; j++)
212             {
213                 if(rewired[i][j] == 1)
214                 {
215                     if(player_strategy[j] == 1)
216                     {
217                         return false;
218                     }
219                 }
220             }
221             // pi out the set
222             else
223             {
224                 int flag = 0;
225                 for(int j = 0; j < 30; j++)
226                 {
227                     if(rewired[i][j] == 1)
228                     {
229                         if(player_strategy[j] == 1)
230                         {
231                             flag = 1;
232                         }
233                     }
234                 }
235                 if(flag == 0)
236                 {
237                     return false;
238                 }
239             }
240         }
241     }
242     return true;
243 }

```

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.

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

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

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.

```

406 //record the matching of node, init the node not in the matching so they are -1
407 int matching[30];
408 for(int i = 0; i < 30; i++)
409     matching[i] = -1;
410
411 // randomize the initial state, the state also a matching
412 int left, right;
413 while(edge > 0)
414 {
415     left = rand() % 30;
416     right = rand() % 30;
417
418     //random select two node which need to connect
419     while(left == right || rewired[left][right] == 0)
420     {
421         left = rand() % 30;
422         right = rand() % 30;
423     }
424
425     // two node need to not in the matching and rand need to = 1 -> then the edge can join to the init matching
426     if(matching[left] != -1 || matching[right] != -1 || rand() % 2 == 0)
427     {
428         edge--;
429         continue;
430     }
431
432     // the edge join to the init matching
433     matching[left] = right;
434     matching[right] = left;
435     edge--;
436 }

```

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.

```

438 while(!NE)
439 {
440     // random pick up one player who can improve its utility
441     pi = rand() % 30;
442     while (player_state[pi] == 1)
443     {
444         pi = rand() % 30;
445     }
446     player_state[pi] = 1;
447
448     // pi in the matching but pi can choose the lower degree neighbor which not in the matching and pair it
449     if(matching[pi] > -1)
450     {
451         int min_degree = degree[matching[pi]];
452         int select_node = matching[pi]; //original pair
453         bool choose = false;
454         for(int i = 0; i < 30; i++)
455         {
456             if(rewired[pi][i] == 1)
457             {
458                 if(degree[i] < min_degree && matching[i] == -1)
459                 {
460                     min_degree = degree[i];
461                     select_node = i;
462                     choose = true;
463                 }
464             }
465         }
466         matching[pi] = select_node;
467         matching[select_node] = pi;
468         if(choose)
469         {
470             move_count++;
471             for (int i = 0; i < 30; i++)
472                 player_state[i] = 0;
473             player_state[pi] = 1;
474         }
475     }

```

```

477 //matching[pi] = -1 , pi not in the matching so pi can choose the min degree node which not in the matching and pair it
478 else
479 {
480     int min_degree = 100;
481     int select_node;
482     bool choose = false;
483     for(int i = 0; i < 30; i++)
484     {
485         if(rewired[pi][i] == 1)
486         {
487             if(degree[i] < min_degree && matching[i] == -1)
488             {
489                 min_degree = degree[i];
490                 select_node = i;
491                 choose = true;
492             }
493         }
494     }
495     if(choose)
496     {
497         matching[pi] = select_node;
498         matching[select_node] = pi;
499         move_count++;
500         for (int i = 0; i < 30; i++)
501         |     player_state[i] = 0;
502         player_state[pi] = 1;
503     }
504 }

```

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.

```

506 // test the all player never change their strategy
507 int flag = 0;
508 for (int i = 0; i < 30; i++)
509 {
510     if (player_state[i] == 0)
511     |     flag = 1;
512 }
513
514 if (flag == 0)
515 |     NE = true;

```

And I calculate the numbers of pair.

```

518 for(int i = 0; i < 30; i++)
519 {
520     if(matching[i] != -1)
521     |     pair++;
522 }
523 pair = pair / 2;

```

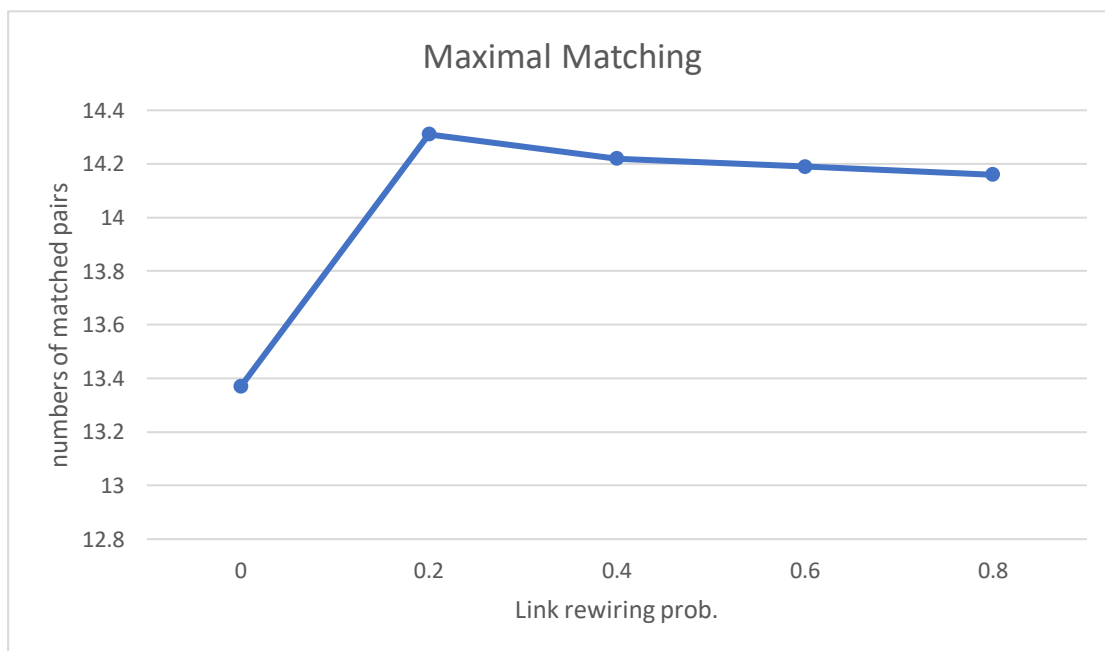
I have a check function to check when reach to Nash Equilibrium, the game state need to satisfy the Maximal Matching game definition. The sample code is as below.

```

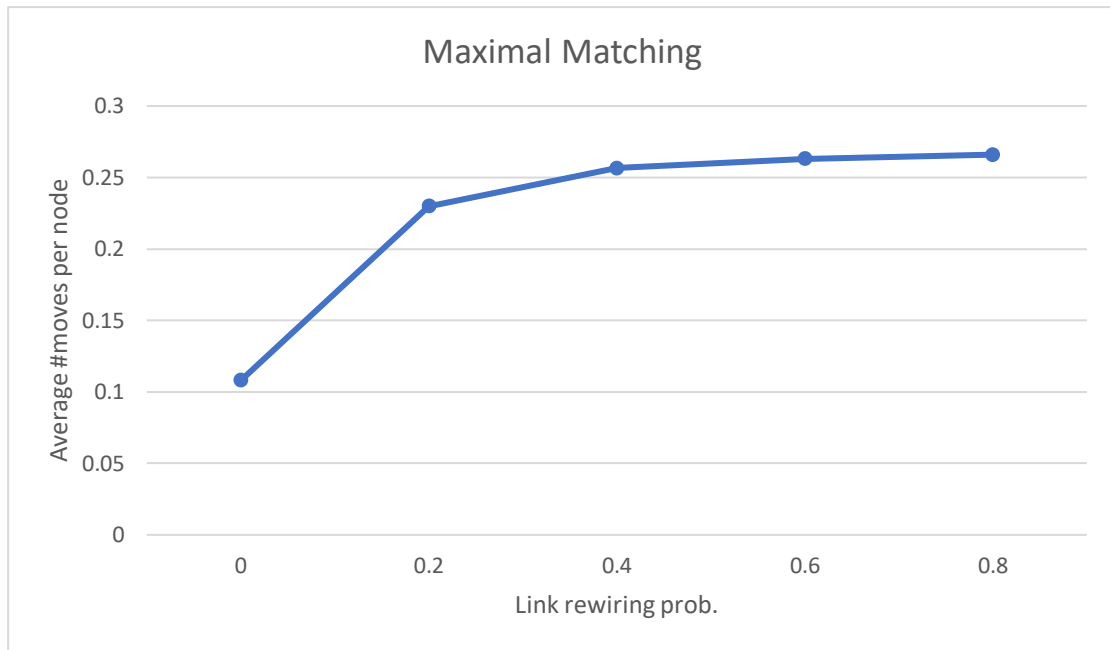
353 bool check_Maximal_Matching(int rewired[30][30], int matching[30])
354 {
355     for(int i = 0; i < 30; i++)
356     {
357         for(int j = 0; j < 30; j++)
358         {
359             if(rewired[i][j] == 1)
360             {
361                 if(matching[i] == j && matching[j] == i)
362                 {
363                     for(int k = 0; k < 30; k++)
364                     {
365                         if(k == i || k == j)
366                             continue;
367                         else
368                             if(matching[k] == i && matching[k] == j)
369                                 return false;
370                     }
371                 }
372             }
373         }
374     }
375     return true;
376 }

```

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.