Iterative Prisoners dilemma

February 26, 2019

1 The Iterative Prisoner's Dilemma

1.0.1 Author : Nisarg Dave

1.0.2 Creation : 17/04/2018

1.1 Introduction

The Game theory is displine of optimizing the Game strategies. It basically examines the conflict scenarios and plots the best possible move for players. The n player games are generally examined by the type of cooperative or non-cooperative games. The zero games where loses to one is the whole winning of the other and the non zero game where the points distribution is different in each case.The game theory breaks down the games first using the number of players,the simultaneous games where players play simultaneously, they do not know what opponant plays.

The representation is done by game winning matrix. The theory focuses on to identify the best way or strategy to win a game and for that it mainly relies on the rationality of the game. One or more rational corrections can be applied.

- The Nash equilibrium , is the situation of no regret for each player. Simultaneous game but now I know what other player played so no regret.
- The optimum of Pareto , The situation where no situation is superior for both the players.

My analysis shows how to compare and apply strategies collectively.

```
In [1]: import numpy as np
        import pandas as pd
        import math
        import itertools
        from random import random
        import random
        import copy
        import datetime
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1.2 A matrix of earnings, Nash equilibrium & pareto

The Game is the class which will allow us to store winnings of both the players. it takes array of pairs (the gain of player 1 and player 2)

Moreover I did use potential balance to calcuate no regret situation corresponding to the nash equilibrium.If the player has 2 max then it's Nash Equilibrium.

```
In [2]: class Game:
            def __init__(self, tab, actions):
                self.actions=actions
                m=np.array(tab,dtype=[('x', 'int32'), ('y', 'int32')])
                self.size = int(math.sqrt(len(tab)))
                self.scores=m.reshape(self.size,self.size)

            def nash(self):
                max_x = np.matrix(self.scores['x'].max(0)).repeat(self.size, axis=0)
                bool_x = self.scores['x'] == max_x
                max_y = np.matrix
                (self.scores['y'].max(1)).transpose().repeat(self.size, axis=1)
                bool_y = self.scores['y'] == max_y
                bool_x_y = bool_x & bool_y
                return self.scores[bool_x_y]

            def isPareto(self, t, s):
                return True if (len(s)==0) else
                (s[0][0] <= t[0] or s[0][1] <= t[1]) and self.isPareto(t, s[1:])

            def pareto(self):
                res = []
                for s in self.scores:
                    if(self.isPareto(s,self.scores)):
                        res.append(s)
                return res
```

```
In [31]: def mutate_population():
             for individual in population:
                 individual.mutate()

         def mutate(self):
                 for i in range(5):
                     if random() < 0.05: #mutation rate
                         self.prbs[i] = random()
```

1.2.1 Nash score

```
In [3]: dip =[(3,3),(0,5),(5,0),(1,1)]    # Dilemma of the prisoner: 1 equilibrium
        gs=[(3,2),(1,1),(0,0),(2,3)]      # War of the sexes: 2 equilibria
        mp=[(1,-1),(-1,1),(-1,1),(1,-1)] # matching pennies : 0 equilibrium
        # paper scissors sheet: 0 equilibrium
```

```
rpc=[(0,0),(-1,1),(1,-1),(1,-1),(0,0),(-1,1),(-1,1),(1,-1),(0,0)]
g = Game(dip,['C','D'])
g.scores
print("The Nash Equilibrium")
g.nash()
#g.pareto()
```

```
The Nash Equilibrium
```

```
Out[3]: array([(1, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
```

1.2.2 Create a random game matrix

```
In [4]: x = np.random.randint(0, 5, (3,3))
        y = np.random.randint(0, 5, (3,3))
        couples = [(a,b) for a,b in zip(x.flatten(),y.flatten())]
        g = Game(couples, None)
        print("The random game matrix scores")
        print(g.scores)
        print("The nash score for those matrix")
        print(g.nash())
```

```
The random game matrix scores
[[(0, 2) (2, 0) (1, 0)]
 [(1, 1) (0, 3) (4, 2)]
 [(3, 3) (1, 2) (1, 2)]]
The nash score for those matrix
[(3, 3)]
```

1.2.3 Generate all matrices

Here is the Cartesian product of values that interests us. We can then for example count how many games have 0.1 or more Nash equilibria.

```
In [5]: def numberOfGames(valeurs, nbCoups):
            return len(valeurs)**((nbCoups**2)*2)

        print("The number of Games generated")
        print(numberOfGames([1,2],2))
```

```
The number of Games generated
256
```

```
In [6]: def enumAllGames(valeurs, nbCoups):
            res = [q for q in itertools.product([p for p in itertools.product
                                                (list(valeurs), repeat=2)],
```

```
                                                    repeat=nbCoups**2)]
            return [[res[j][k] for k in range(nbCoups**2)] for j in range(len(res))]

        n = enumAllGames([1,2],2)
        print("Print 10 random games found out of "+str(numberOfGames([1,2],2)))
        for i in range (10):
            print(random.choice(n))
```

```
Print 10 random games found out of 256
[(1, 1), (2, 2), (2, 2), (1, 1)]
[(1, 1), (1, 1), (2, 1), (2, 2)]
[(1, 2), (1, 1), (1, 2), (1, 2)]
[(1, 2), (2, 1), (2, 2), (1, 2)]
[(2, 1), (2, 1), (2, 1), (2, 1)]
[(2, 1), (2, 1), (1, 1), (2, 2)]
[(2, 2), (2, 2), (2, 1), (1, 1)]
[(2, 1), (1, 2), (1, 1), (1, 2)]
[(1, 1), (2, 1), (1, 1), (1, 2)]
[(1, 2), (1, 2), (2, 1), (2, 1)]
```

1.2.4 Getting nash equilibrium for the same

```
In [7]: def countNashEquilibria(valeurs, coups):
            results = [Game(i,None).nash().size
                       for i in enumAllGames(valeurs, coups)]
            return dict((i,results.count(i)) for i in set(results))

        # How many 2-beat games on (1,2) on x Nash equilibrium
        print("Nash equilibrium is")
        countNashEquilibria([1,2],2)
```

```
Nash equilibrium is


Out[7]: {0: 2, 1: 44, 2: 114, 3: 80, 4: 16}
```

1.3 Strategy

The startegy decides the next move in game. In terms of this environment, Information available is the move matrix. Let's create the strategy class.

```
In [8]: class Strategy():
            def setMemory(self,mem):
                pass

            def getAction(self,tick):
                pass
```

```
    def __copy__(self):
        pass

    def update(self,x,y):
        pass
```

1.4 Periodic version of strategy

```
In [9]: class Periodic(Strategy):
            def __init__(self, sequence, name=None):
                super().__init__()
                self.sequence = sequence.upper()
                self.step = 0
                self.name = "per_"+sequence if (name == None) else name

            def getAction(self,tick):
                return self.sequence[tick % len(self.sequence)]

            def clone(self):
                object = Periodic(self.sequence, self.name)
                return object

            def update(self,x,y):
                pass
```

1.4.1 Creating example of periodic strategy

```
In [10]: s1 = Periodic("abc")
         print(s1.name,end="\t")
         for i in range (0,10):
             print(s1.getAction(i), end=' ')
         # IT MUST HAVE 10 MOVES. STARTS WITH AND IS FINISHED BY A
```

```
per_abc        A B C A B C A B C A
```

1.5 Strategy meeting

According to game matrix, when two strategy meets during certain number of moves, the score of each is sum of the scores.

```
In [11]: class Meeting :
             def __init__(self,game,s1,s2,length=1000):
                 self.game = game
                 self.s1=s1.clone()
                 self.s2=s2.clone()
                 self.length=length

             def reinit(self):
                 self.s1_score=0
```

```
            self.s2_score=0
            self.s1_rounds=""
            self.s2_rounds=""

        def run(self):
            self.reinit()
            for tick in range(0,self.length):
                c1=self.s1.getAction(tick).upper()
                c2=self.s2.getAction(tick).upper()
                self.s1_rounds+=c1
                self.s2_rounds+=c2
                self.s1.update(c1,c2)
                self.s2.update(c2,c1)
                act=self.game.actions
                self.s1_score+=self.game.scores['x'][act.index(c1),act.index(c2)]
                self.s2_score+=self.game.scores['y'][act.index(c1),act.index(c2)]
```

1.5.1 Evaluating periodic strategy

```
In [12]: dip =[(3,3),(0,5),(5,0),(1,1)]    # Dilemma of the prisoner
         g = Game(dip,['C','D'])
         s1=Periodic("CCD")
         s2=Periodic("DDC")
         m = Meeting(g,s1,s2,10)
         m.run()
         print(m.s1.name+"\t"+m.s1_rounds+" "+str(m.s1_score))
         print(m.s2.name+"\t"+m.s2_rounds+" "+str(m.s2_score))
         # ON MUST HAVE 15.35
```

```
per_CCD        CCDCCDCCDC 15
per_DDC        DDCDDCDDCD 35
```

1.6 Tournament

In the tournament, using two different strategies. The winning strategy gets the highest score

```
In [13]: class Tournament:
             def __init__(self, game, strategies, length=1000, repeat=1):
                 self.strategies = strategies
                 self.game = game
                 self.length=length
                 self.repeat=repeat
                 size=len(strategies);
                 df = pd.DataFrame(np.zeros((size,size+1),dtype=np.int32))
                 df.columns, df.index = [s.name for s in self.strategies]+
                 ["Total"], [s.name for s in self.strategies]
                 self.matrix = df
```

```
    def run(self):
        for k in range(self.repeat):
            for i in range (0,len(self.strategies)):
                for j in range (i,len(self.strategies)):
                    meet = Meeting(self.game, self.strategies[i],
                                   self.strategies[j], self.length)
                    meet.run()
                    self.matrix.at[self.strategies[i].name,
                                   self.strategies[j].name] = meet.s1_score
                    self.matrix.at[self.strategies[j].name,
                                   self.strategies[i].name] = meet.s2_score
        self.matrix["Total"] = self.matrix.sum(axis=1)
        self.matrix.sort_values(by='Total', ascending=False, inplace=True)
        rows = list(self.matrix.index) + ["Total"]
        self.matrix = self.matrix.reindex(columns=rows)
```

1.7 Playing tournament using those strategies

In [14]:
```
bag = []
bag.append(Periodic('C'))
bag.append(Periodic('D'))
bag.append(Periodic('DDC'))
bag.append(Periodic('CCD'))
t=Tournament(g,bag,10)
t.run()
print(t.matrix)
# ON 10 SHOTS: [('per_D', 120), ('per_DDC', 102), ('per_CCD', 78), ('per_C', 60)]
```

```
         per_D  per_DDC  per_CCD  per_C  Total
per_D       10       22       38     50    120
per_DDC      7       16       35     44    102
per_CCD      3       15       24     36     78
per_C        0        9       21     30     60
```

1.8 Generate sets of Strategies

In [15]:
```
cards = ['C','D']
periodics = [p for p in itertools.product(cards, repeat=1)]+
            [p for p in itertools.product(cards, repeat=2)] +
            [p for p in itertools.product(cards, repeat=3)]
strats = [Periodic(''.join(p)) for p in periodics] # join to transform in strings
print(str(len(strats))+" Generated Strategies")
# 14 ARE GENERATED: 2 to one shot, 4 to two shots, 8 to three shots
```

```
14 Generated Strategies
```

1.9 Ecological Competitions

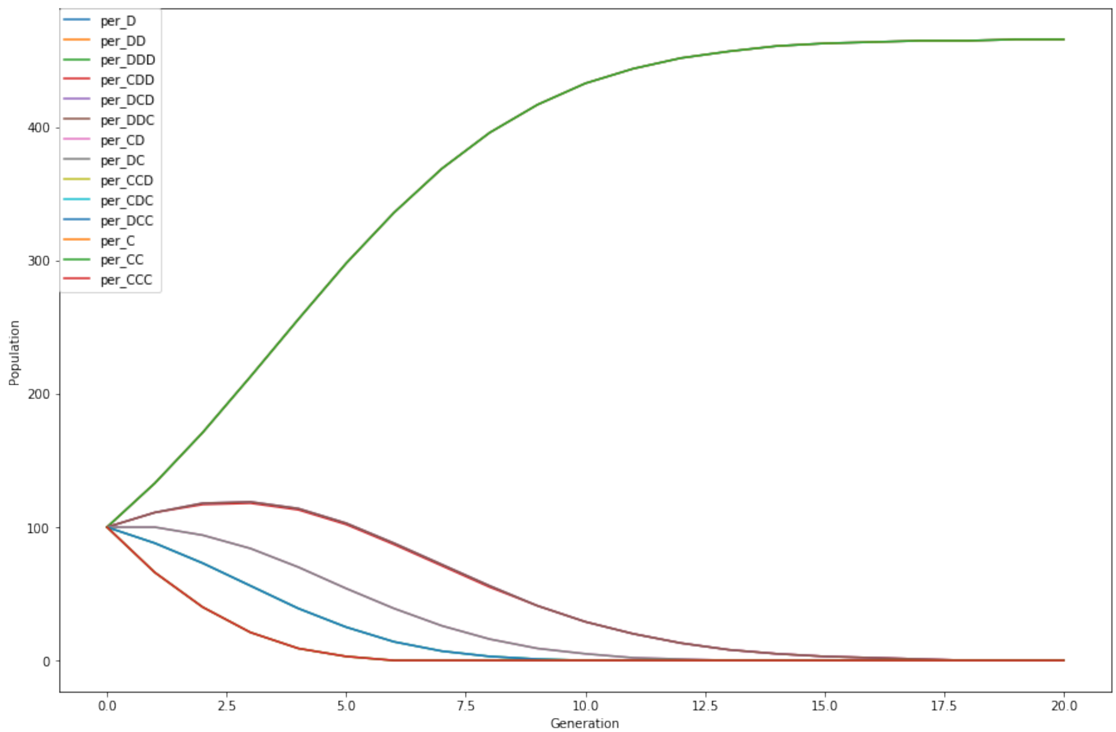
There is a tournament with n game matrix. The each of them are playing with each other. Step 1: The representatives are obtained according to their success. Step 2: Establishing ecological ranking is robust and drawing the graph generations vs population.

```
In [16]: class Ecological:
             def __init__(self, game, strategies, length=1000, repeat=1, pop=100):
                 self.strategies = strategies
                 self.pop = pop
                 self.game = game
                 self.generation = 0 # Number of the current generation
                 self.base = pop*len(strategies)
                 self.historic = pd.DataFrame(columns =
                                           [strat.name for strat in strategies])
                 self.historic.loc[0] = [pop for x in range (len(strategies))]
                 self.extinctions = dict((s.name,math.inf) for s in strategies)
                 self.scores = dict((s.name,0) for s in strategies)
                 self.tournament = Tournament(self.game, self.strategies,length,repeat)
                 self.tournament.run()

             def run(self):
                 stab = False
                 while ((self.generation < 1000) and (stab==False)):
                     parents = list(copy.copy(self.historic.loc[self.generation]))
                     for i in range (len(self.strategies)):
                         strat=self.strategies[i].name
                         score = 0
                         for j in range(len(self.strategies)):
                             strat2 = self.strategies[j].name
                             if i==j:
                                 score+=(self.historic.at[self.generation,
                                                          strat]-1)*
                                 self.tournament.matrix.at[strat,strat2]
                             else:
                                 score+=self.historic.at[self.generation,
                                                         strat2]*
                                 self.tournament.matrix.at[strat,strat2]
                         self.scores[strat] = score
                     total = 0
                     for strat in self.strategies:
                         total+=self.scores[strat.name]*
                         self.historic.at[self.generation, strat.name]
                     for strat in self.strategies:
                         parent = self.historic.at[self.generation, strat.name]
                         self.historic.at[self.generation+1, strat.name] =
                         math.floor(self.base*parent*self.scores[strat.name]/total)
                         if ((parent!=0) and (self.historic.at[
                             self.generation+1, strat.name] == 0)):
```

```
                    self.extinctions[strat.name] = self.generation+1
                elif (self.historic.at[self.generation+1, strat.name] != 0):
                    self.extinctions[strat.name] =
                    elf.historic.at[self.generation+1, strat.name]*1000
            self.generation+=1
            if (parents == list(self.historic.loc[self.generation])):stab = True
        trie = sorted(self.extinctions.items(), key=lambda t:t[1], reverse=True)
        df_trie = pd.DataFrame()
        for t in trie :
            df_trie[t[0]]=self.historic[t[0]]
        self.historic = df_trie
        return self.historic

    def saveData(self):
        date = datetime.datetime.now()
        self.historic.to_csv(str(date)+'.csv', sep=';', encoding='utf-8')

    def drawPlot(self,nbCourbes=None,nbLegends=None):
        nbCourbes = len(self.strategies) if (nbCourbes==None) else nbCourbes
        nbLegends = len(self.strategies) if (nbLegends==None) else nbLegends
        strat = self.historic.columns.tolist()
        for i in range(nbCourbes):
            plt.plot(self.historic[strat[i]], label=strat[i]
                     if (i<nbLegends) else '_nolegend_')
        plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
        plt.ylabel('Population')
        plt.xlabel('Generation')
        plt.show()
        date = datetime.datetime.now()
        plt.savefig(str(date)+'.png', dpi=1000)
```

1.9.1 Plotting generations vs population

In [17]:
```
eco = Ecological(g, strats)
eco.run()
plt.figure(figsize=(15,10))    # to set the size of the figure
eco.drawPlot(None,None)
eco.historic
```



Out[17]:

	per_D	per_DD	per_DDD	per_CDD	per_DCD	per_DDC	per_CD	per_DC	per_CCD	per_CDC \
0	100	100	100	100	100	100	100	100	100	100
1	133	133	133	111	111	111	100	100	88	88
2	171	171	171	117	118	118	94	94	73	73
3	213	213	213	118	119	119	84	84	56	56
4	256	256	256	113	114	114	70	70	39	39
5	298	298	298	102	103	103	54	54	25	25
6	336	336	336	87	88	88	39	39	14	14
7	369	369	369	71	72	72	26	26	7	7
8	396	396	396	55	56	56	16	16	3	3
9	417	417	417	41	41	41	9	9	1	1
10	433	433	433	29	29	29	5	5	0	0
11	444	444	444	20	20	20	2	2	0	0
12	452	452	452	13	13	13	1	1	0	0
13	457	457	457	8	8	8	0	0	0	0
14	461	461	461	5	5	5	0	0	0	0
15	463	463	463	3	3	3	0	0	0	0
16	464	464	464	2	2	2	0	0	0	0
17	465	465	465	1	1	1	0	0	0	0
18	465	465	465	0	0	0	0	0	0	0
19	466	466	466	0	0	0	0	0	0	0
20	466	466	466	0	0	0	0	0	0	0

per_DCC per_C per_CC per_CCC

```
0      100   100   100   100
1       88    66    66    66
2       73    40    40    40
3       56    21    21    21
4       39     9     9     9
5       25     3     3     3
6       14     0     0     0
7        7     0     0     0
8        3     0     0     0
9        1     0     0     0
10       0     0     0     0
11       0     0     0     0
12       0     0     0     0
13       0     0     0     0
14       0     0     0     0
15       0     0     0     0
16       0     0     0     0
17       0     0     0     0
18       0     0     0     0
19       0     0     0     0
20       0     0     0     0
```

```
<Figure size 432x288 with 0 Axes>
```

1.10 Reactive strategies

1.11 Tit for tat strategy: cooperates on the first move then plays what its opponent played the previous move.

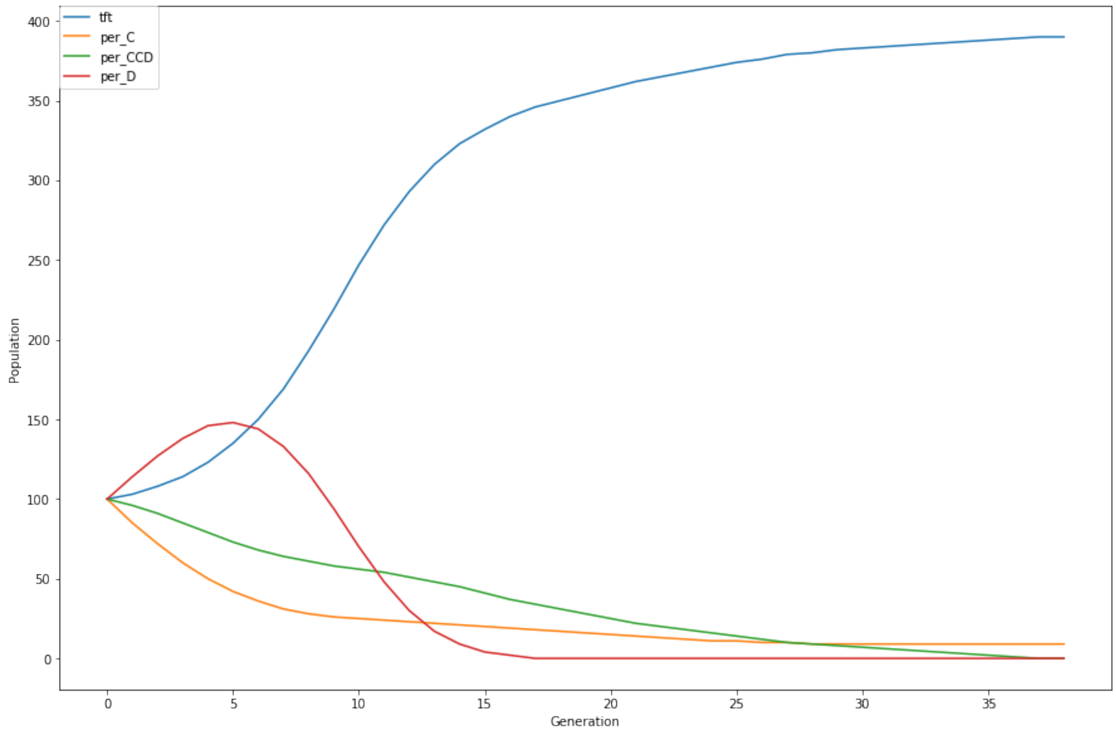
```
In [18]: class Tft(Strategy):
             def __init__(self):
                 super().__init__()
                 self.name = "tft"
                 self.hisPast=""

             def getAction(self,tick):
                 return 'C' if (tick==0) else self.hisPast[-1]

             def clone(self):
                 return Tft()

             def update(self,my,his):
                 self.hisPast+=his
```

1.12 Spiteful: cooperates until the opponent defects and thereafter always defects.

```
In [19]: class Spiteful(Strategy):
             def __init__(self):
```

```
        super().__init__()
        self.name = "spiteful"
        self.hisPast=""
        self.myPast=""

    def getAction(self,tick):
        if (tick==0):
                return 'C'
        if (self.hisPast[-1]=='D' or self.myPast[-1]=='D') :
            return 'D'
        else :
            return 'C'

    def clone(self):
        return Spiteful()

    def update(self,my,his):
        self.myPast+=my
        self.hisPast+=his
```

In [20]:
```
eco = Ecological(g, [Periodic('C'),Periodic('D'),Periodic('CCD'),Tft()])
eco.run()
plt.figure(figsize=(15,10))    # to set the size of the figure
eco.drawPlot()
eco.tournament.matrix
plt.savefig("toto.png")
eco.historic
# IN THIS EXPERIENCE, ALL_D WINS THE TOURNAMENT,
#BUT IT'S TFT WINNING THE ECOLOGICAL COMPETITION
```



Out[20]:

	tft	per_C	per_CCD	per_D
0	100	100	100	100
1	103	85	96	114
2	108	72	91	127
3	114	60	85	138
4	123	50	79	146
5	135	42	73	148
6	150	36	68	144
7	169	31	64	133
8	193	28	61	116
9	219	26	58	94
10	247	25	56	70
11	272	24	54	48
12	293	23	51	30
13	310	22	48	17
14	323	21	45	9
15	332	20	41	4
16	340	19	37	2
17	346	18	34	0
18	350	17	31	0
19	354	16	28	0
20	358	15	25	0
21	362	14	22	0
22	365	13	20	0

```
23  368    12      18      0
24  371    11      16      0
25  374    11      14      0
26  376    10      12      0
27  379    10      10      0
28  380     9       9      0
29  382     9       8      0
30  383     9       7      0
31  384     9       6      0
32  385     9       5      0
33  386     9       4      0
34  387     9       3      0
35  388     9       2      0
36  389     9       1      0
37  390     9       0      0
38  390     9       0      0
```

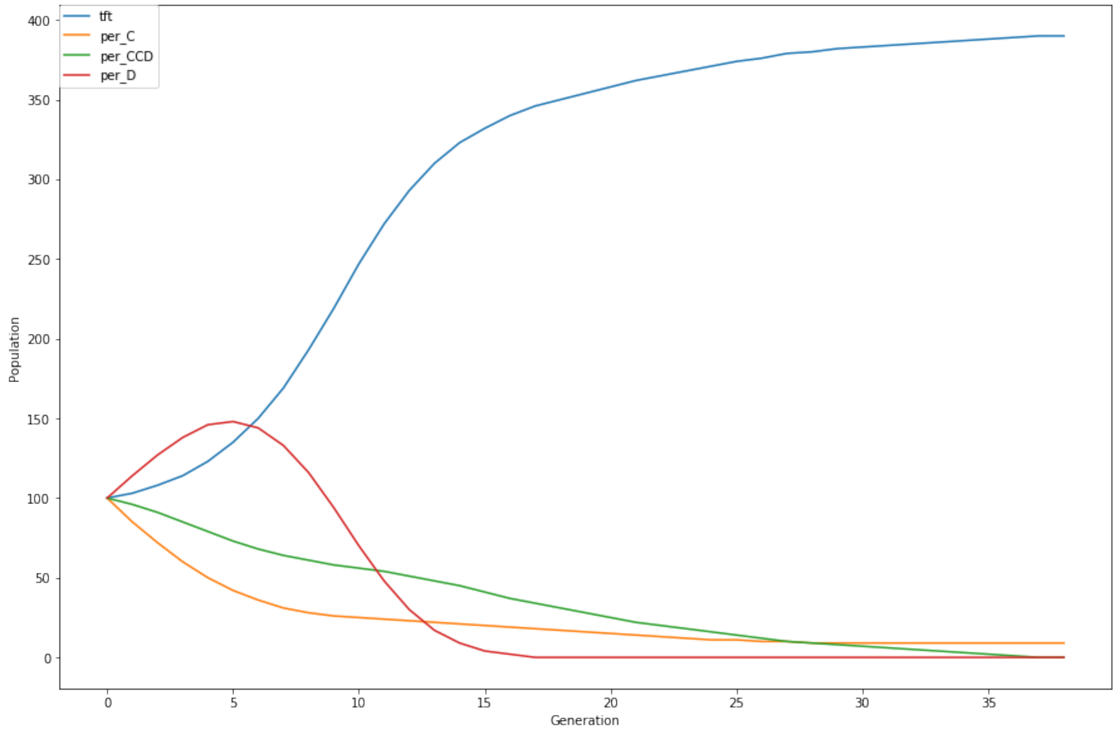
```
<Figure size 432x288 with 0 Axes>
```

```
In [21]: tournament = eco.tournament.matrix

         print ("--- The complete matrix of the sorted tournament")
         print (tournament)
         print ("--- The winners of the tournament")
         print (tournament [ 'Total'])
         print ("--- The first 3 winners of the tournament")
         print (tournament [ 'Total'] [0: 3])
         print ("--- Winners who made more than 10,000")
         print (tournament [ 'Total'] [tournament [ 'Total']> 10000])

         evol = eco.historic

         print ("--- The complete sorted history")
         print (evol)
         print ("--- Final populations ranked")
         print (evol.iloc [eco.generation])
         print (evol.iloc [-1])
         print (evol.tail (1))
         print ("--- The first 2 of the competition")
         print (evol.iloc [-1] [0: 2])
         print ("--- The last survivors")
         print (evol.iloc [-1] [evol.iloc [-1]> 0])
         print ("--- the line when tft = 340?")
         evol.loc [evol.tft == 340]
         print ("--- What index do per_C and per_D intersect?")
         print (evol.loc [evol.per_C> evol.per_D].loc [evol.per_D != 0])
```

```python
#Write the equivalent of select * from evol where ...

evol.loc [(evol.tft> 300) & (evol.per_D> 0)]
evol.query ('tft> 300 & per_D> 0')
plt.figure(figsize=(15,10))    # to set the size of the figure
eco.drawPlot()
```

```
--- The complete matrix of the sorted tournament
         per_D   tft  per_CCD  per_C  Total
per_D     1000  1004     3668   5000  10672
tft        999  3000     2667   3000   9666
per_CCD    333  2667     2334   3666   9000
per_C        0  3000     2001   3000   8001
--- The winners of the tournament
per_D      10672
tft         9666
per_CCD     9000
per_C       8001
Name: Total, dtype: int64
--- The first 3 winners of the tournament
per_D      10672
tft         9666
per_CCD     9000
Name: Total, dtype: int64
--- Winners who made more than 10,000
per_D    10672
Name: Total, dtype: int64
--- The complete sorted history
    tft per_C per_CCD per_D
0   100   100     100   100
1   103    85      96   114
2   108    72      91   127
3   114    60      85   138
4   123    50      79   146
5   135    42      73   148
6   150    36      68   144
7   169    31      64   133
8   193    28      61   116
9   219    26      58    94
10  247    25      56    70
11  272    24      54    48
12  293    23      51    30
13  310    22      48    17
14  323    21      45     9
15  332    20      41     4
16  340    19      37     2
17  346    18      34     0
18  350    17      31     0
```

```
19  354    16      28     0
20  358    15      25     0
21  362    14      22     0
22  365    13      20     0
23  368    12      18     0
24  371    11      16     0
25  374    11      14     0
26  376    10      12     0
27  379    10      10     0
28  380     9       9     0
29  382     9       8     0
30  383     9       7     0
31  384     9       6     0
32  385     9       5     0
33  386     9       4     0
34  387     9       3     0
35  388     9       2     0
36  389     9       1     0
37  390     9       0     0
38  390     9       0     0
--- Final populations ranked
tft        390
per_C        9
per_CCD      0
per_D        0
Name: 38, dtype: object
tft        390
per_C        9
per_CCD      0
per_D        0
Name: 38, dtype: object
    tft per_C per_CCD per_D
38  390     9       0     0
--- The first 2 of the competition
tft      390
per_C      9
Name: 38, dtype: object
--- The last survivors
tft      390
per_C      9
Name: 38, dtype: object
--- the line when tft = 340?
--- What index do per_C and per_D intersect?
    tft per_C per_CCD per_D
13  310    22      48    17
14  323    21      45     9
15  332    20      41     4
16  340    19      37     2
```




```
<Figure size 432x288 with 0 Axes>
```

2 The n -memory based IPD

```
In [22]: class Mem(Strategy):
             def __init__(self, x, y, genome, name=None):
                 self.name = name
                 self.x = x
                 self.y = y
                 self.genome = genome
                 if (name == None):# Default name if the user does not define it
                     self.name = genome
                 self.myMoves = [] #contains my x last moves
                 self.itsMoves = [] #contains its y last moves

             def clone(self):
                 return Mem(self.x, self.y, self.genome, self.name)

             def getAction(self, tick):
                 if (tick < max(self.x, self.y)):
```

```
                return self.genome[tick]
            cpt = 0
            for i in range(self.x-1,-1,-1):
                cpt*=2
                if (self.myMoves[i] == 'D'):
                    cpt+=1
            for i in range(self.y-1,-1,-1):
                cpt*=2
                if (self.itsMoves[i] == 'D'):
                    cpt+=1
            cpt += max(self.x, self.y)
            return self.genome[cpt]

        def update(self, myMove, itsMove):
            if (self.x > 0):
                if(len(self.myMoves) == self.x):
                    del self.myMoves[0]
                self.myMoves.append(myMove)
            if (self.y > 0):
                if(len(self.itsMoves) == self.y):
                    del self.itsMoves[0]
                self.itsMoves.append(itsMove)
```

2.1 Memory based strategies

1) Always Cooperate : Cooperates on every move
2) Always Defect : Defects on every move3) Tit for Tat : Cooperates on the first move, then simply copies the opponent's last move.
3) Pavlov : Cooperates on the first move, and defects only if both the players did not agree on the previous move.
4) Spiteful : Cooperates, until the opponent defects, and thereafter always defects.
5) Random Player : Makes a random move.
6) Periodic player CD : Plays C, D periodically.
7) Periodic player DDC : Plays D, D, C periodically.
8) Periodic player CCD : Plays C, C, D periodically.
9) Tit for Two Tats : Cooperates on the first move, and defects only when the opponent defects two times.
10) Hard Tit for Tat : Cooperates on the first move, and defects if the opponent has defects on any of the previous three moves, else cooperates.

```
In [23]: Mem(0,0,'C','allc')
         Mem(0,0,'D','alld')
         Mem(1,0,'cDC','percd')
         Mem(1,0,'dDC','perdc')
         Mem(0,1,'cCD','tft')
         Mem(0,1,'dCD','mistrust')
         Mem(1,1,'cCDDD','spiteful')
         Mem(1,1,'cCDDC','pavlov')
```

```
Mem(0,2,'ccCCCD','tf2t')
Mem(0,2,'ccCDDD','hard_tft')
Mem(1,2,'ccCCCDCDDD','slow_tft')
Mem(1,2,'ccCDCDDCDD','winner12')
Mem(1,2,'','tft_spiteful')
Mem(1,2,'ccCDDDDDDD','spiteful_cc')
```

Out[23]: <__main__.Mem at 0x7f16ebbf1390>

In [24]:
```
bag1 = [Periodic('C'),Periodic('D'),Tft(),Spiteful(),Periodic('CD'),Periodic('DC')]
t1=Tournament(g,bag1,100)
t1.run()
print(t1.matrix)

bag2 = [Mem(0,0,'C','allc'),Mem(0,0,'D','alld'),Mem(0,1,'cCD','tft'),
        Mem(1,1,'cCDDD','spiteful'),Mem(1,0,'cDC','percd'),Mem(1,0,'dDC','perdc')]
t2=Tournament(g,bag2,100)
t2.run()
print(t2.matrix)
```

```
          spiteful  tft  per_D  per_CD  per_DC  per_C  Total
spiteful       300  300     99     297     299    300   1595
tft            300  300     99     248     250    300   1497
per_D          104  104    100     300     300    500   1408
per_CD          57  253     50     200     250    400   1210
per_DC          54  250     50     250     200    400   1204
per_C          300  300      0     150     150    300   1200
          spiteful  tft  alld  percd  perdc  allc  Total
spiteful       300  300    99    297    299   300   1595
tft            300  300    99    248    250   300   1497
alld           104  104   100    300    300   500   1408
percd           57  253    50    200    250   400   1210
perdc           54  250    50    250    200   400   1204
allc           300  300     0    150    150   300   1200
```

2.2 Generate all

Family | Length | First Game |
:-: :-: |
mem(0,1) | 1+2^1 = 3 | 2^3 = 8 |
mem(1,0) | 1+2^1 = 3 | 2^3 = 8 |
mem(1,1) | 1+2^2 = 5 | 2^5 = 32 |
mem(2,0) | 2+2^2 = 6 | 2^6 = 64 |
mem(1,2) | 2+2^3 = 10 | 2^10 = 1024 |
mem(2,1) | 2+2^3 = 10 | 2^10 = 1024 |
mem(2,2) | 2+2^4 = 18 | 2^18 = 262144 |

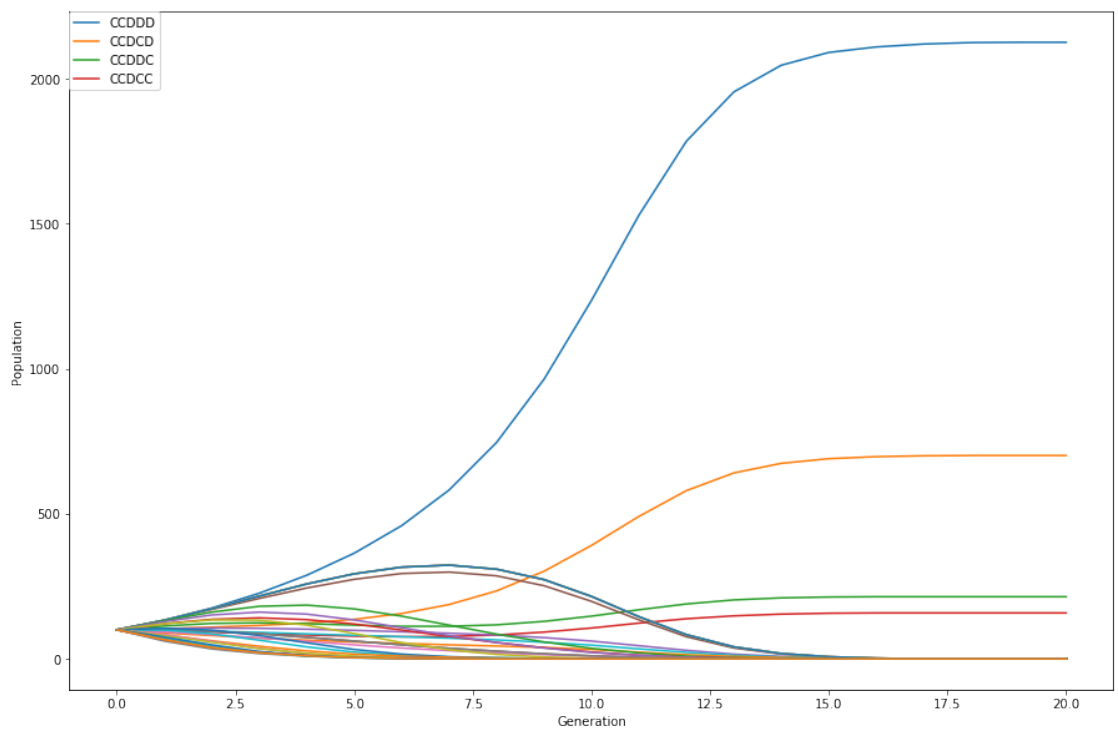
In [25]:
```
def getAllMemory(x,y):
    if (x+y > 4):
```

```
            return "Pas calculable"
        len_genome = max(x,y)+2**(x+y)
        permut = [p for p in itertools.product(['C','D'], repeat=len_genome)]
        genomes = [''.join(p) for p in permut]
        return [Mem(x,y,gen) for gen in genomes]

print("In Mem (1,1) there is "+ str(len(getAllMemory(1,1))) + " stratégies")
```

```
In Mem (1,1) there is 32 stratégies
```

2.3 The competition of Mem(1,1)

```
In [26]: bag3 = getAllMemory(1,1)
         e2 = Ecological(g,bag3)
         e2.run()
         plt.figure(figsize=(15,10))
         e2.drawPlot(None,4)
         evol = e2.historic
         print(evol.iloc[-1][evol.iloc[-1]>0])
         # Only 4 survive: mem11_cCDDD-spite 2126, mem11_cCDCD-tft 701,
         #mem11_cCDDD-pavlov 214, mem11_cCDCC 158
```




```
CCDDD     2126
CCDCD      701
```

```
CCDDC      214
CCDCC      158
Name: 20, dtype: object
```

```
<Figure size 432x288 with 0 Axes>
```

2.4 Beating the one memory class

find a `Mem (2,2)` capable of winning in the set of `Mem (1,1)`. The simplest way to find one is to calculate a random genotype of a `Mem (2,2)` a number of times, evaluate it in `Mem (1,1)`, look at its ranking and keep only the one with the highest ranking. This is the case of `Mem (2,2, 'CCCCDDDCCDCDDDDDDD')`.

```
In [27]: class FindBest:
             def __init__(self, game):
                 self.game = game

             def generate_random_genotype(self, x, y):
                 N = max(x,y) + 2**(x+y)
                 genotype = ""
                 for i in range (N):
                     genotype += random.choice(self.game.actions)
                 return genotype

             def random_selection(self, x, y, nb_tests, soupe):
                 d = dict()
                 for n in range(nb_tests):
                     genotype = self.generate_random_genotype(x,y)
                     strat = Mem(x,y,genotype)
                     eco = Ecological(self.game, soupe+[strat])
                     eco.run()
                     d[genotype] = eco.historic.columns.tolist().index(strat.name)
                 return sorted(d.items(), key=lambda t: t[1])
```

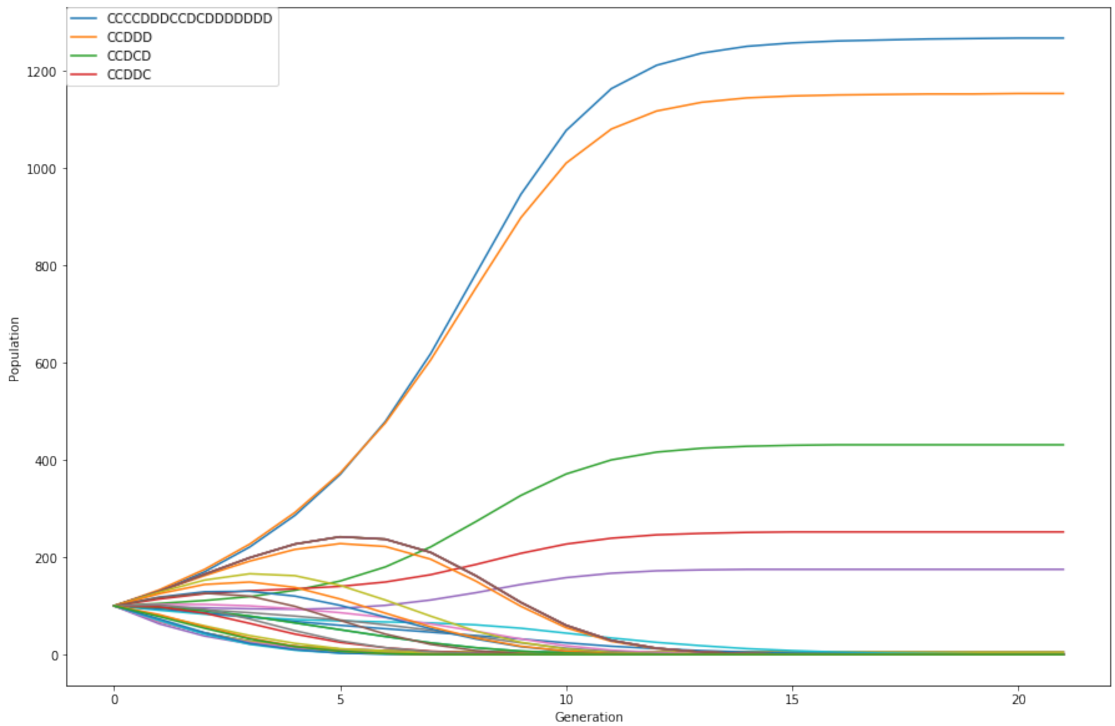
```
In [28]: dip =[(3,3),(0,5),(5,0),(1,1)]
         g = Game(dip,['C','D'])
         gen = FindBest(g)
         soupe = getAllMemory(1,1)
         print(gen.random_selection(2,2,10,soupe)[1:3])
```

```
[('DCDCCDCCCCCDDDCDCD', 12), ('CDCDDCDDDDDDDCCDDC', 15)]
```

2.5 Plotting the generations vs Population

```
In [29]: bag3=getAllMemory(1,1)
         e2=Ecological(g,bag3+[Mem(2,2,'CCCCDDDCCDCDDDDDDD')])
```

```python
e2.run()
plt.figure(figsize=(15,10))
e2.drawPlot(None,4)
evol=e2.historic
print(evol.iloc[-1][evol.iloc[-1]>0])
```




```
CCCCDDDCCDCDDDDDDD    1267
CCDDD                 1153
CCDCD                  431
CCDDC                  252
CCDCC                  175
CCCCC                    5
CCCCD                    5
CCCDC                    5
CCCDD                    5
Name: 21, dtype: object
```

```
<Figure size 432x288 with 0 Axes>
```

3 Conclusion: This project was fun. I loved reading on Game theory and Nash equilibirum. I love the concept of No regret for anyone and finding the strategy which can compensate both.