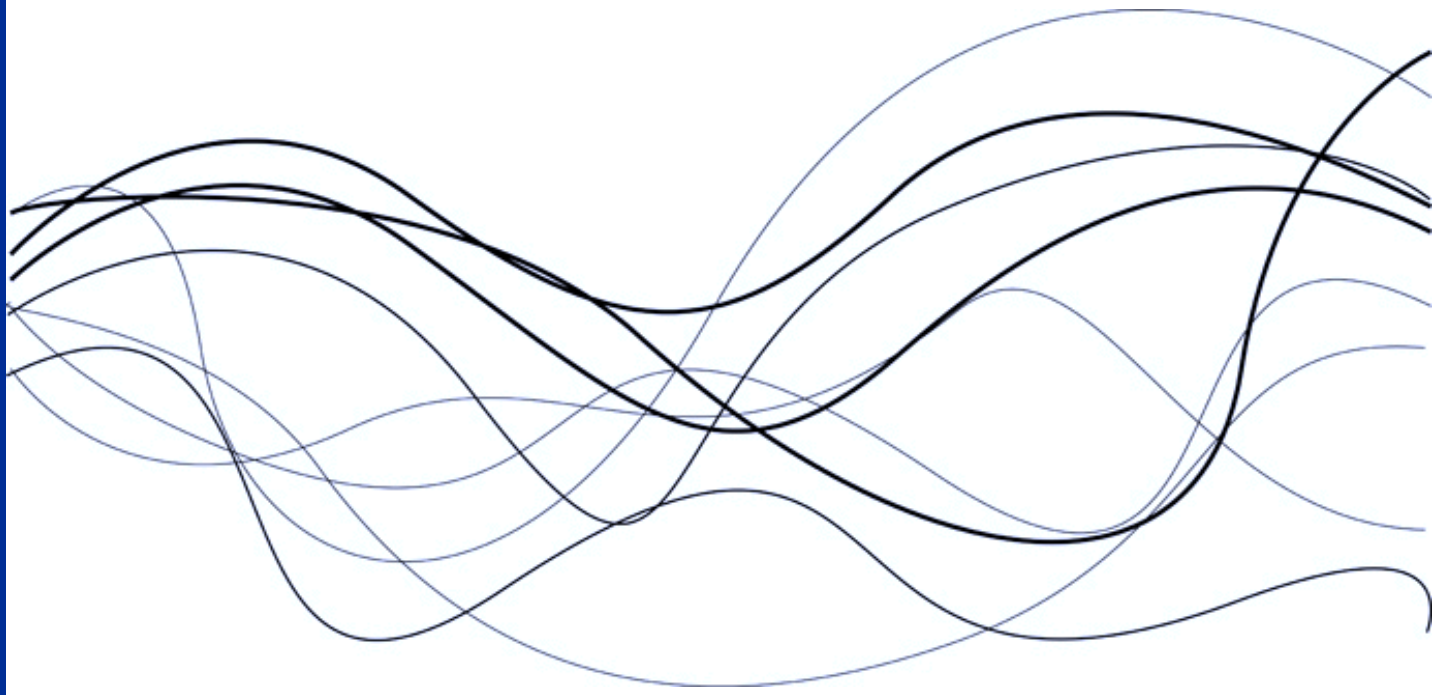
MAY 18, 2019



ANALYSIS OF THE ITERATED PRISONER'S DILEMMA

LAUREN FOSSEL
PHIL 126: PARADOXES
Professor McPartland

INTRODUCTION .

You and your friend Ted are arrested for robbing a jewelry shop. Neither of you is a great runner, so the police catch you both and bring you to be interrogated in separate rooms. You are given the following deal:

1. If you defect (betray Ted) but he cooperates (stays silent), you will go free and Ted will get 5 years in jail.
2. If Ted defects you but you cooperate, Ted will go free and you will get 5 years in jail.
3. If both of you defect, you both will get 3 years in jail.
4. If both of you cooperate, you both will get 1 year in jail.

From the outline above, it appears that the most logical choice for you is to defect, because no matter what Ted chooses, you will personally be better off by betraying him than staying silent. Assuming first that Ted cooperates, if you also cooperate, you'll get 1 year in jail, but if you defect, you'll get 0 years, so clearly you're better off defecting if Ted cooperates. If Ted defects and you cooperate, you'll get 5 years in jail, but if you also defect, you'll only get 2 years. 2 years in jail is better than 5, so clearly you are better defecting. This is, of course, assuming that you are only trying to minimize your prison sentence (it does not take into account, for example, guilt or getting beaten up for being a snitch).

So, it appears that no matter what Ted does, you'll be better off if you defect. However, what if you really like your jewelry, and you know that after your prison sentence, you'll want to steal more? And what if you know that you'll want Ted as you partner in crime again? Or if he isn't your partner, you'd want Ned, but Ted and Ned are friends, so Ted will tell Ned if you betray

him, then what? If one plays more than a single round of the prisoner's dilemma (the so-called iterated prisoner's dilemma), more complicated strategies can emerge that take into account the future implications of current choices.

AXELROD'S TOURNAMENT

Robert Axelrod, a political scientist at the University of Michigan, conducted two prisoner's dilemma tournaments around 1980. For the first, Axelrod gathered strategies from other game theorists to play 200 rounds of the iterated prisoner's dilemma. Results were tallied by summing the total points each strategy gained throughout the tournament. His second tournament was more complicated and is of more interest to us. This tournament was a sort of "evolutionary" prisoner's dilemma, where the prevalence of each strategy in the next round was determined by that particular strategy's success in the previous round. This is essentially the idea of evolution and survival of the fittest: the strategies that do the best reproduce and therefore take up a larger percentage of the population in the future than the strategies that don't do as well. For both these tournaments, the dominant strategy turned out to be Tit for tat, which is explained below.

STRATEGIES

- The Cooperator: this strategy cooperates no matter what, meaning that it always chooses to stay silent.
- The Defector: this strategy defects no matter what, meaning that it always chooses to betray the other person.
- Tit For Tat: this strategy cooperates on the first round, and for every round after, it copies what the other player did on the previous round. It is a retaliating, yet also

forgiving strategy, meaning that if you defect against a Tit for Tat, it will retaliate and defect against you. However, if you then cooperate, it will forgive you and also cooperate.

- Majority Determined Tit For Tat: this strategy knows all of its opponent's previous plays, and will choose to play what its opponent most commonly played.
- Percentage Tit For Tat: this strategy, similar to the one above, knows all of its opponent's previous moves, but uses the percent the opponent plays either cooperate or defect to determine its move. For example, if its opponent cooperates 75% of the time, it would also cooperate 75% of the time. Though there is a random element to this strategy, it still mimics the other player's moves.
- Random: this strategy chooses whether to cooperate or defect randomly. As implemented below, it can be given a percentage such that it will cooperate that percent of the time. When given the percent 100, it would be equivalent to a Cooperator. When give the percent 0, it would be equivalent to a Defector.

BASIC IMPLEMENTATION

When designing this version of the ecological or evolutionary prisoner's dilemma, we had to make some key decisions about the implementation. First, we decided to have a constant number of players. Whatever the initial number of players was, that would stay constant (the ratio of each strategy could of course change throughout the evolution). This makes sense in terms of an ecosystem in the real world. An ecosystem only has a finite amount of resources, so only a finite number of animals can live in it. The species that are better at using the resources, or, perhaps,

better at fighting off the other species, will be the ones that survive to reproduce. As Darwin said hundreds of years ago and which has now become a common saying: survival of the fittest.

The second key decision about implementation was how to determine the number of players each strategy would have after each round. For each strategy x, the implementation we decided on was:

$$\frac{\text{total points for each x in previous round}}{\text{total points earned in previous round}} * (\#total\ players\ in\ previous\ round)$$

This means that the number of players of each strategy in the next round was determined by the total number of points earned by the strategy in the previous round. This simulates how if an initial population of an unsuccessful strategy is introduced to an ecosystem, after time the strategy will die out. It also simulates that even if a successful strategy doesn't have very many players to begin with, its success means that it could eventually increase its number of players. Note that players were randomly matched for each game (which could, of course, have multiple rounds). Multiple rounds of choosing defect or cooperate make up a game, and multiple games make up a tournament (or evolutionary round). Players have the same opponents for each round of a game, but different opponents for each game in a tournament.

The final key choice of implementation had to do with the Random strategy. A dilemma emerged when trying to determine how the random strategy should evolve through multiple rounds. If each Random player can have its own percentage chance of choosing to cooperate, how should new Random players be "born" through evolution and which players should die when the random population shrinks? We decided that if the Random strategy will increase its number of players in the next round, all the current players (and their percentages) will continue on, and the

Random players that got the most points will create more Random players with their same percentages. This helps follow the idea that the members of the species that are most successful will pass on their genes more often than the ones that are less successful. If the Random strategy will decrease its number of players in the next round, the Random players with the most points will continue on, and the ones with the least points will die off. Though this could have also been solved by having each individual player have its own descendants, the purpose of this program was to look at populations of strategies as a whole. Plus, the advantage of doing specific descendants (having the more successful Random players populate more than less successful ones) is achieved using the outline above.

EQUAL INITIAL POPULATION EVOLUTION

Figure 1 displays the results of starting each heuristic's population at 30, having 30 rounds to each individual game, and having 100 games per tournament. Each line corresponds an average of 5 tournament runs. As seen in the figure, Defectors died off first, followed shortly by Random players (playing with a 50% probability). Though the Cooperators never died out completely, of the heuristics that survived, they did the worst and ended with fewer players than they started with. The Tit for Tat, Majority Determined Tit For Tat, and Percentage Tit For Tat all did the best and were in the same region. Though the Defectors, Cooperators, and Random Players all had fairly consistent results, the various Tit For Tats had differing results over the various runs, though always high. This helps illustrate that though the Tit for Tats are more successful, they are also less stable and depend more on which other types of players they get matched with.

After averaging the final number of players over 100 trials of 100 rounds of total evolution (that is, instead of running 100 games 5 times as displayed in the Figure 1, running the total tournement a total of 100 times, still with 30 rounds per game), the final populations were as follows (in increasing order):

- Defectors: 0.00
- Random (50/50) 0.00
- Cooperators: 20.80
- Majority Tit For Tat: 48.58
- Tit For Tat: 51.93
- Percentage Tit For Tat: 52.11

Clearly, the 3 types of Tit For Tats were much more successful than the other players, and all had very similar results. The percent difference between the Majority Tit For Tat and Percentage Tit For Tat is only around 7%. This makes sense because they all have the same basic concept: punish the other player as much as they punish you.

ANALYSIS OF THE DIFFERENT RANDOMS

Figure 2 below shows the Random players playing with different percentages of cooperating. It shows each of the Random player's population throughout the evolution, with each heuristic (Cooperate, Defect, Tit For Tat, Random, Majority Tit For Tat, and Percentage Tit For Tat) starting with 30 players each. Note that each evolution was only played with Random players of like percentages (that is, 30% Random players never played with 70% Random players and so on). The players with a 10-40% percentage had decreasing populations immediately. This makes

sense because they are essentially Defectors that occasionally cooperate, and Defectors are generally unsuccessful and go extinct quickly.

The Random players with 60-90% initially increased their population, reached a peak, and then started decreasing. This makes sense because initially the Random players with high probability of cooperating can trick the "cleverer" players (the Majority Tit For Tat and Percent Tit for Tat) into thinking that they always cooperate. When the Random players defect instead of cooperate, they gain points against the Majority Tit For Tat and Percent Tit For Tat. Though not displayed in the figure, the regular Tit For Tat and Percentage Tit for Tat are the most successful against these Random players because they can't be tricked into thinking that a Random player will always cooperate. So, though in an evolutionary game with 50/50 Random players the three Tit For Tats essentially do equally as well, when Random players with other percentages are introduced, the regular Tit For Tat and Percentage Tit for Tat are the most successful because they cannot be tricked like the Majority Tit for Tat can.

ANALOGY TO REPUTATION

The Tit for Tats, along with the Random players, have illuminating relations to the importance of reputation. The Majority Tit for Tat can be thought of a person who knows that another player tends to play one move more than the other, so it will always play its opponent's most common move. The idea is that the Majority Tit for Tat player would like to maintain their own reputation of cooperating, however, if it knows its opponent tends to defect, it will defect too. The Majority Tit for Tat player can be easily taken advantage of, however, because all the other players have to do is maintain a cooperation rate of 50% or greater, and then play defect every so often, and

they will win against the Majority Tit for Tat. This can be interpreted to mean that a player just has to maintain a reputation for cooperating more than they defect, and then defect occasionally, and they will win. This is reflected in figure 3b.

The Percent Tit for Tats and regular Tit for Tats are much more successful against Random players than the Majority Tit for Tat is, because they essentially are more skeptical of reputation. Both of these Tit for Tats mirror their opponent's cooperation ratio, so if they find that their opponent defects 25% of the time, they will also defect 25% of the time. This means that these Tit for Tats win against Random players in the evolution simulation, because even though they tie when they play against the Random players, when the respective Tit for Tats play against each other, they never defect, and hence get more points. This is reflected in figures 3a and 3c below.

A MINORITY OF DEFECTORS

Imagine a population with a large majority of Cooperators and Tit for Tats, and a small population of Defectors. The question is whether these Defectors will be able to take advantage of the Cooperators and Tit for Tats. The Defectors will always win against the Cooperators, and will at least tie with the Tit for Tats, so it seems that the Defectors would do well in this population. However, the Cooperators and Tit for Tats do very well with each other, whereas the Defectors don't do well against other Defectors.

Figure 4 shows the average number of Defectors alive after each round of evolution based on the initial population of Defectors. The simulation was run with initially 100 Cooperators, 100 Tit

for Tats, and various initial numbers of Defectors (4, 8, 12, 16, 20), and each tournament had 40 rounds of games (each game had 30 rounds each). As seen in the graph, though the Defectors' populations may fluctuate, they generally tend towards 0.

Figure 5 displays the results after running the same simulation a total of 500 rounds of evolution for each initial number of Defectors (2, 4, 8, 12, 16, 20, 24, 28, 32, 36). It shows the average number of rounds the Defectors survived to until the population became 0. The longer the Defectors survive, the more success they have as a strategy. As seen in the graph, the Defectors' success initially increases as the initial population number increases, then reaches a peak at an initial population of 12, and then starts dying off as the initial population gets too large. This makes sense because with too few initial Defectors they can't get enough points to survive, but with too many Defectors, there's more of a chance they'll play against each other, in which case they will get far fewer points than the Cooperators and Tit for Tats. This simulation illustrates that the best initial percentage population for Defectors in a population of Cooperators and Tit for Tats is 3%. Though even with 3% the Defectors will die off, with more or less than 3% the Defectors will die off much faster.

CONCLUSIONS

Starting from the concept of an evolutionary prisoner's dilemma, we looked in depth at different possible scenarios. We saw that against a diverse population, all the Tit for Tats do approximately the same, but solely against Random players, the regular Tit for Tat and Percentage Tit for Tat do the best, as they cannot be tricked by reputation like the Majority Tit for Tat can. We saw that the most successful Random players were the ones that almost always

cooperate, but occasionally defect. We also saw that Defectors are most successful in a population of Cooperators and Tit for Tats when the Defectors make up 3% of the population.

These results showed us some important takeaways: 1) reputation is important, but one should have accurate and constantly updated information about their opponent's reputation in order to not get tricked. 2) If you are not going to play Tit for Tat but don't want to just defect or just cooperate, it is best to almost always cooperate and occasionally defect (like the high percentage Random players). You will overall do well against Tit for Tats, and can take advantage of Cooperators while still maintaining a good reputation with the Majority and Percent Tit for Tats. 3) "Mean" strategies (Defectors) do best when they have a small minority of the population, so that they can take advantage of the "nice" strategies (Cooperators and Tit for Tats) without having to play against each other.

An interesting extension of this project would be to look at specific descendants of players instead of having each strategy as a whole have a certain number of descendants. This would require a completely different format for the computer program, and I suspect would best be done in a more complex programming language such as Java. With this format, one could look more in depth into reputation; we could imagine that a player, instead of being omniscient in regard to each player's previous moves as the Percentage Tit for Tat player was, only knew information about a player's previous moves if they had specifically played against each other before. We could imagine a simulation involving degrees of separation and confidence about reputation: if you know someone who's played with your current opponent, perhaps you have some idea of what they tend to play, but you are not as confident as if you had played against

them yourself before. There are an endless number of different scenarios, extensions, and hypotheses that can be explored with the iterated prisoner's dilemma. So, come cooperate and explore them.

Figure 1:

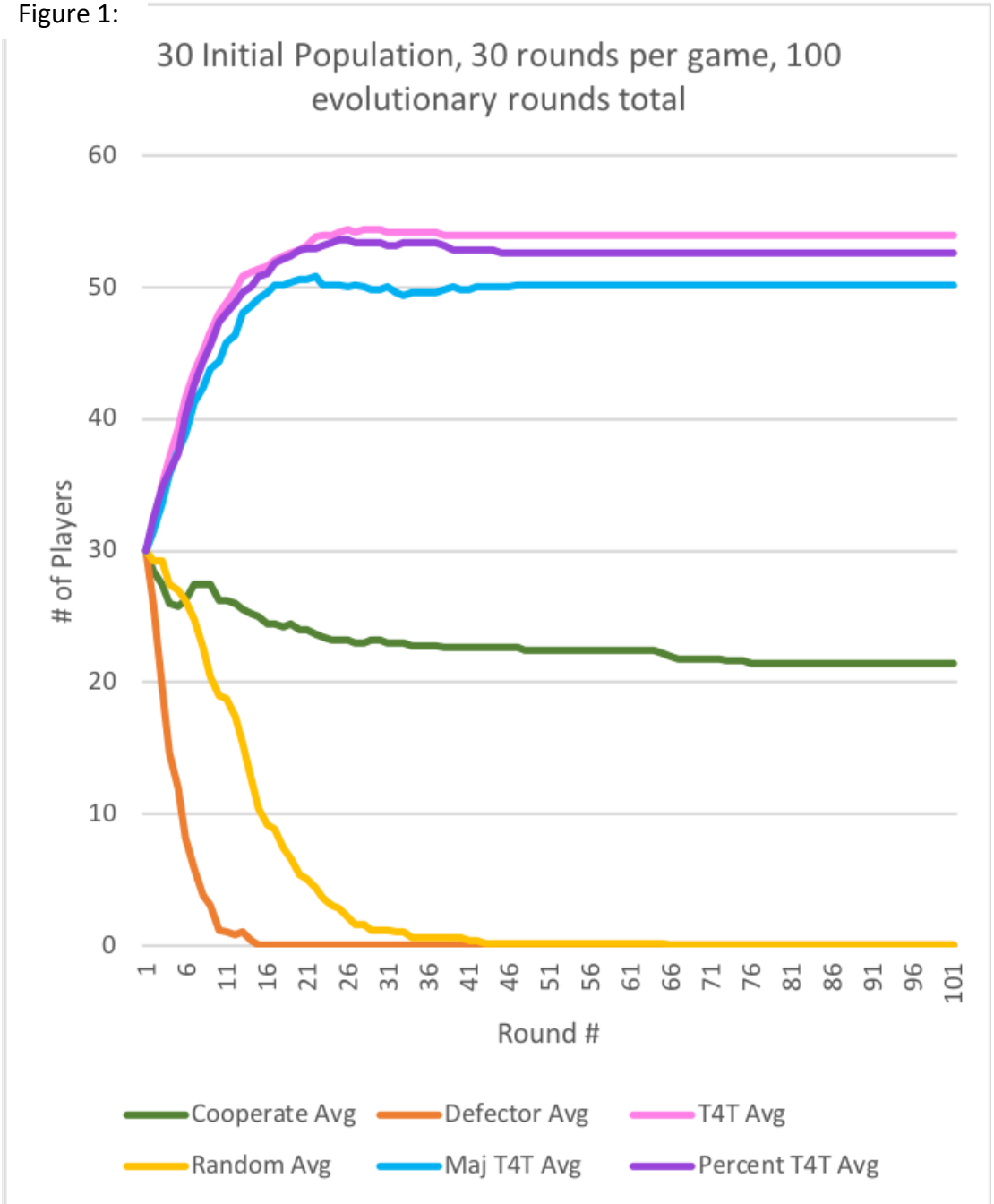


Figure 2:

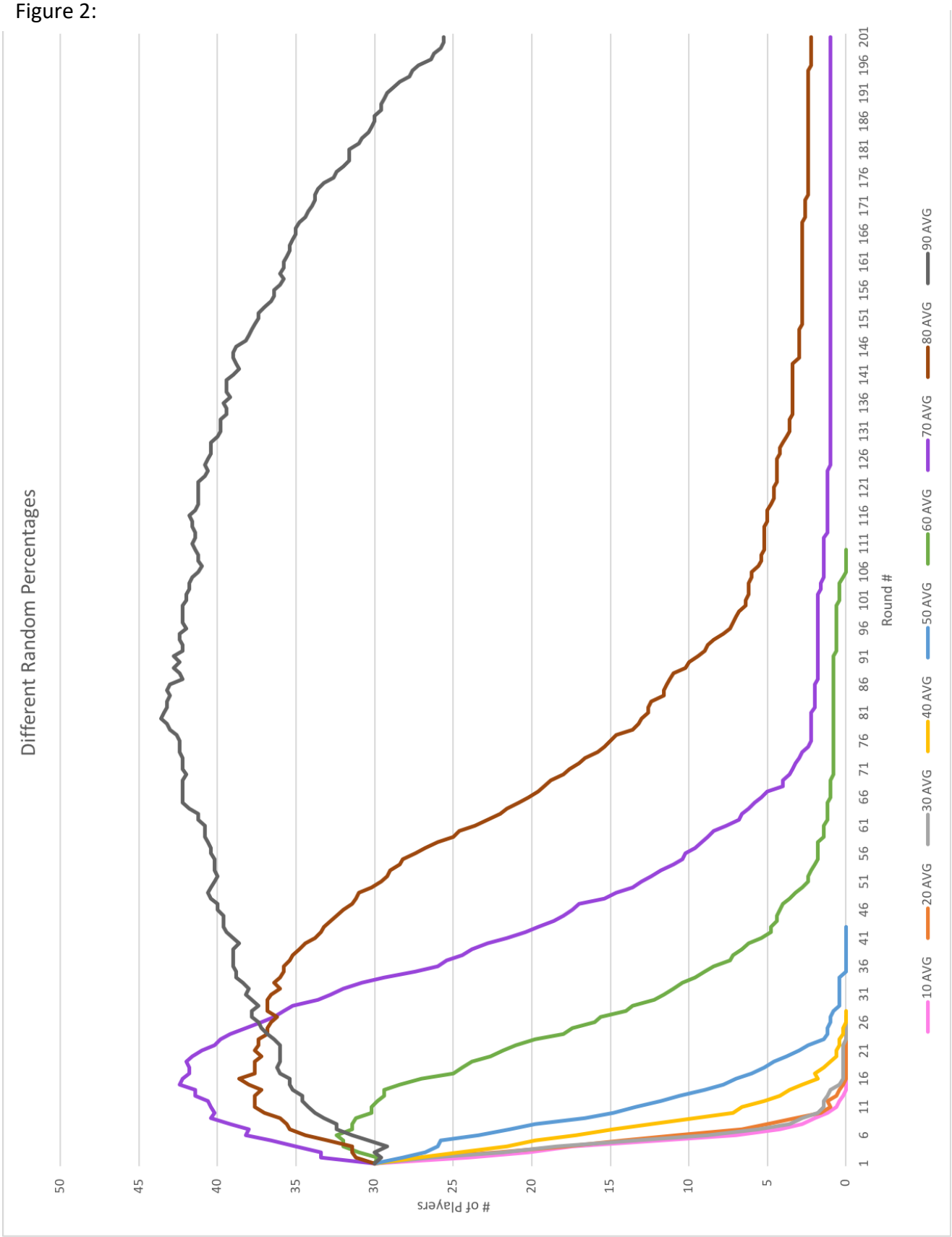

Different Random Percentages
# of Players
Round #
10 AVG
20 AVG
30 AVG
40 AVG
50 AVG
60 AVG
70 AVG
80 AVG
90 AVG

Figure 3a:

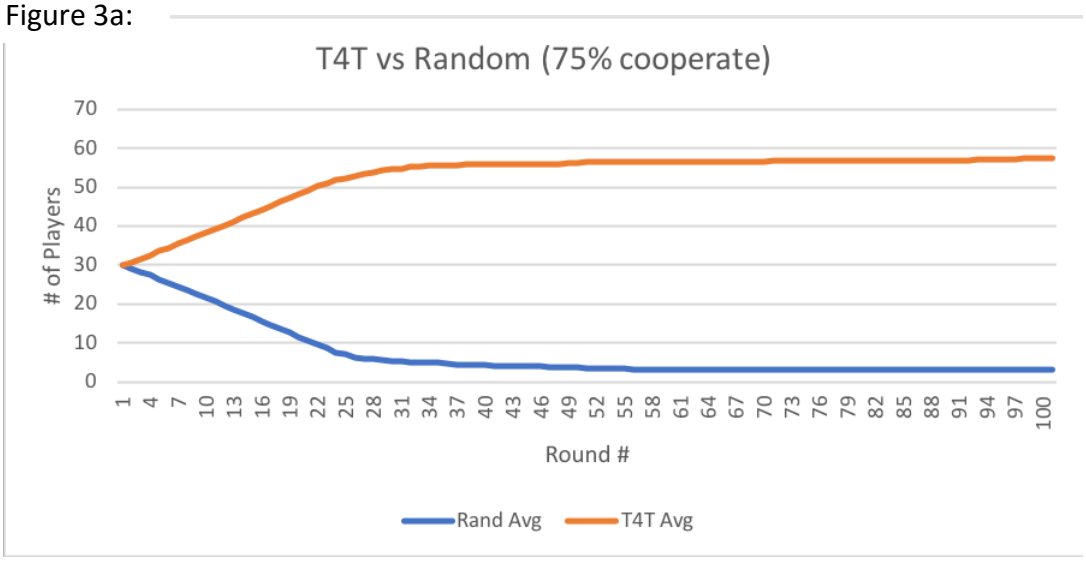


Figure 3b:

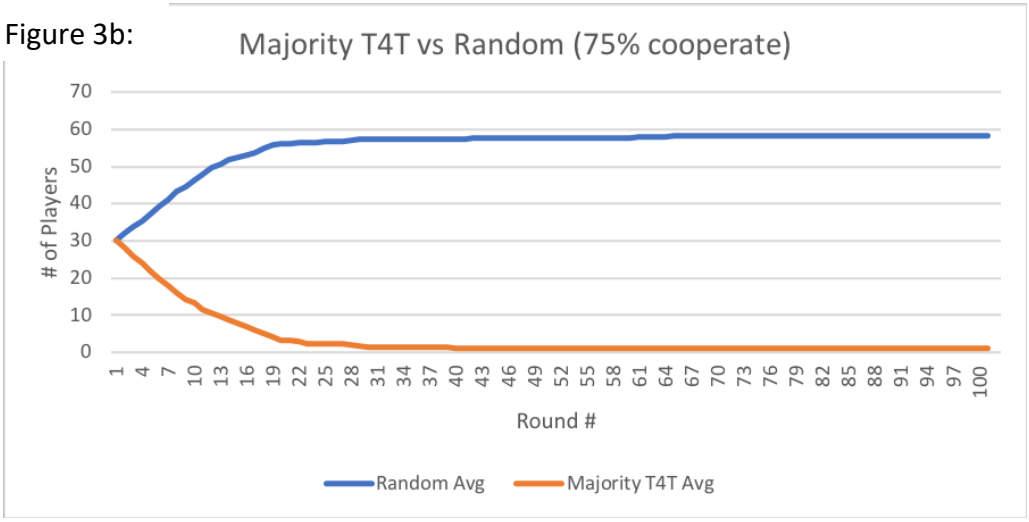


Figure 3c:

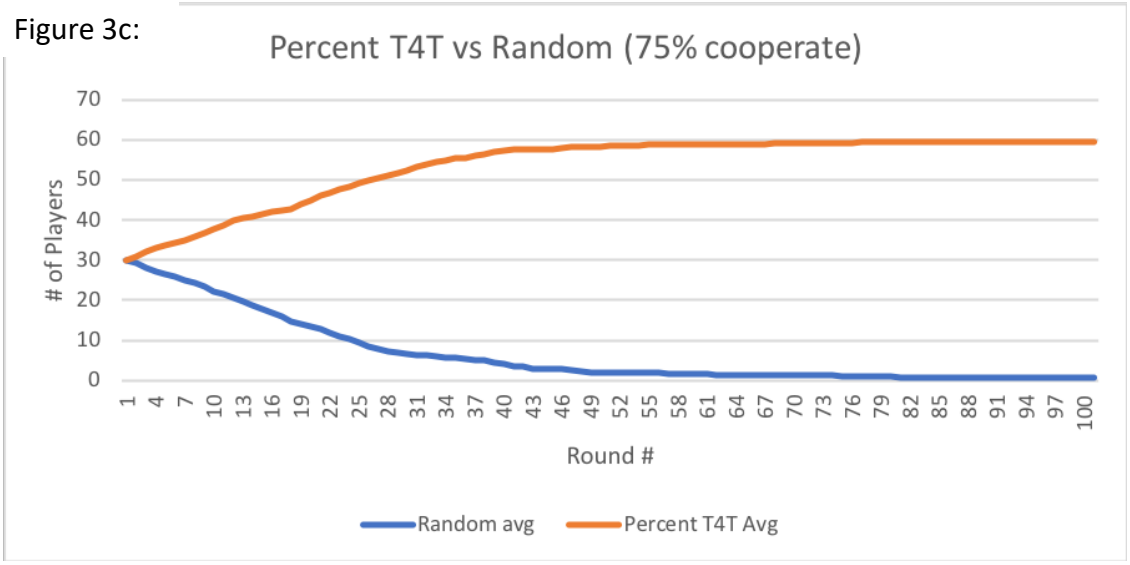


Figure 4:

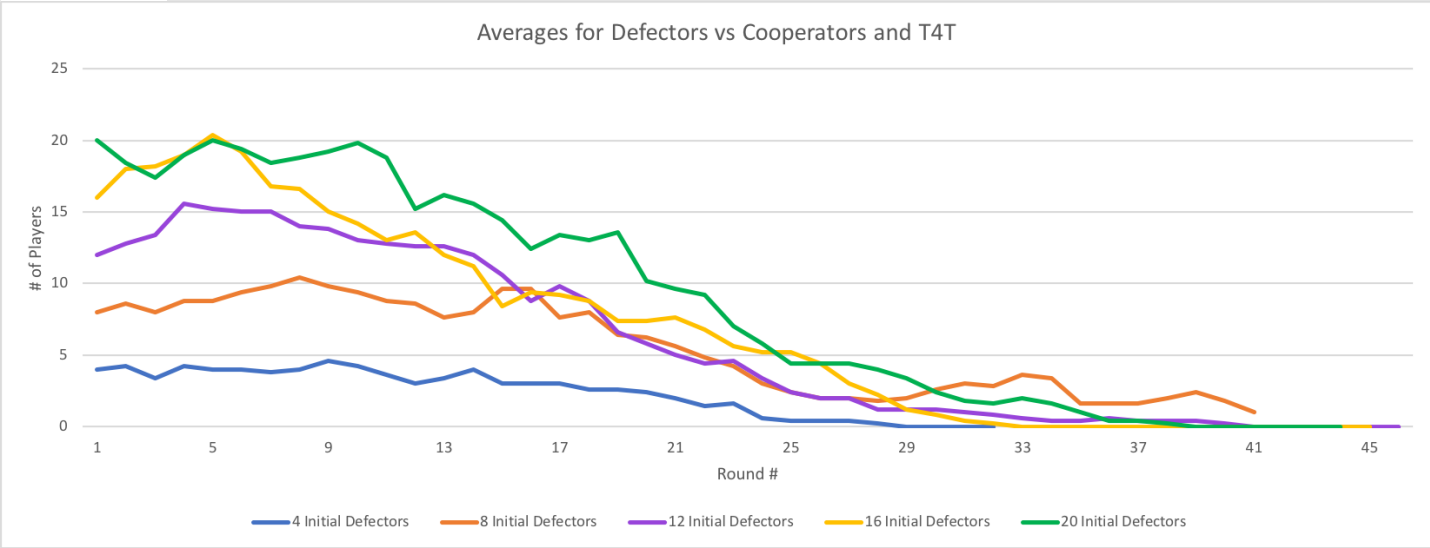
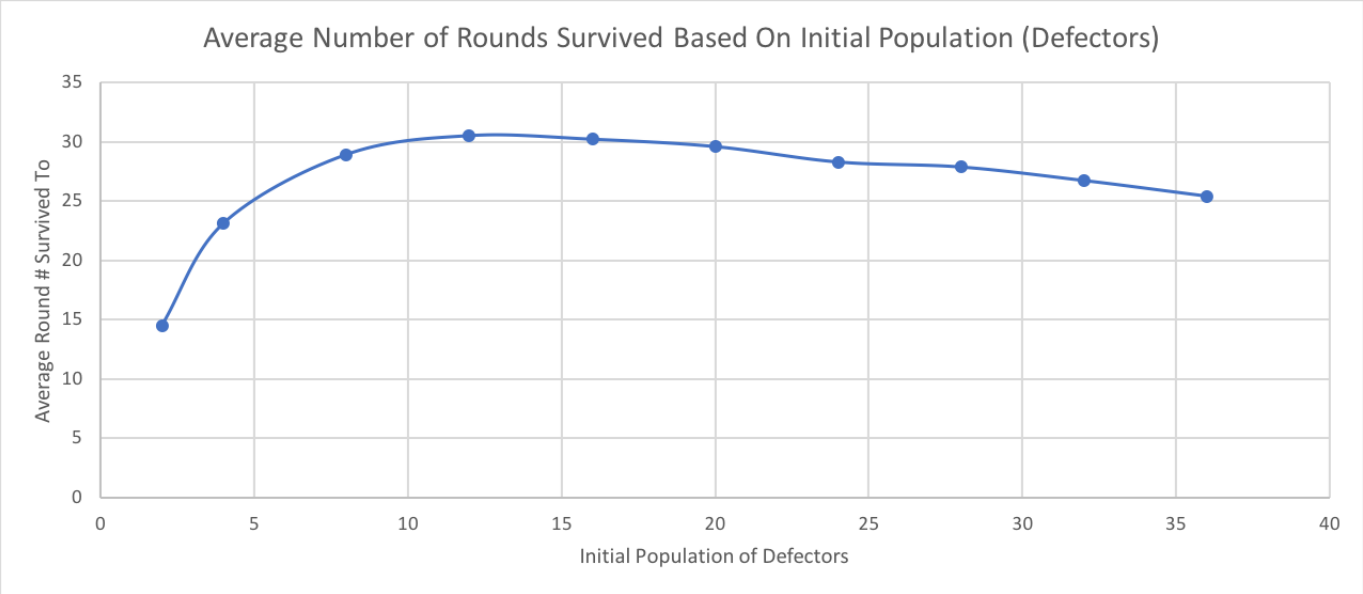


Figure 5:



Code:

```
''' (c) 2019 Kyle Cox and Lauren Fossel
This script simulates an iterated Prisoner's Dilemma Game, inspired by
Axelrod's Tournament.
Expriments with different heuristics:
Coop: cooperates every time
Defect: defects every time
Rand: chooses to defect or cooperate randomly
T4T: tit for tat cooperates first round, then mimics opponent's last move
every other iteration
'''
#MajorityRep
#RandomRep
from __future__ import print_function
from random import randint
from random import choice
from itertools import combinations
from random import shuffle
from random import random
import sys

def matrix(d1, d2):
        # takes 2 ints as matrix indices (p1 move, p2, move)
        # returns tup of matrix outcome
        m = [[(3,3), (0,5)],
            [(5,0), (1,1)]]
        # tup[0] = p1 payoff, tup[1] = p2 payoff
        # goal is higher point value
        # [[(C, C), (C, D)],
        # [(D, C), (D, D)]]
        return m[d1][d2]
class Coop(object):
    # Coop always cooperates (return 0)
    def __init__(self):
        self.name = 'Cooperator'
        self.plays = []
        self.opp_plays = []
        self.round = 0
        self.sum = 0

    def play(self):
        move = 0
        self.plays.append(move)
        self.round += 1
        return move
class Defect(object):
    # Defect always defects (return 1)
    def __init__(self):
        self.name = 'Defector'
        self.plays = []
        self.opp_plays = []
        self.round = 0
        self.sum = 0
```

```
    def play(self):
        self.round += 1
        move = 1
        self.plays.append(move)
        return move

class T4T(object):
    # T4T cooperates first move, then returns the opp's last move for the
rest of the game
    def __init__(self):
        self.name = 'Tit For Tat'
        self.plays = list()
        self.opp_plays = list()
        self.round = 0
        self.sum = 0
    def play(self):
        self.round += 1
        if self.round == 1:
            move = 0
            self.plays.append(move)
            return move
        else:
            move = self.opp_plays[self.round - 2]
            self.plays.append(move)
            return move
class RandomRep(object):
    # cooperates first move, then returns a random element from opp's moves
    def __init__(self):
        self.name = 'Random Reputation'
        self.plays = []
        self.opp_plays = []
        self.round = 0
        self.sum = 0

    def play(self):
        self.round += 1
        if self.round == 1:
            move = 0
            self.plays.append(move)
            return move
        else:
            move = choice(self.opp_plays[:self.round - 1])
            self.plays.append(move)
            return move
class MajorityRep(object):
    # cooperates first move, then if opp is majority defector, defects, if
opp is majority cooperator, coops
    def __init__(self):
        self.name = 'Majority Reputation'
        self.plays = []
        self.opp_plays = []
        self.round = 0
        self.sum = 0

    def play(self):
```

```
        self.round += 1
        if self.round == 1:
            move = 0
            self.plays.append(move)
            return move
        else:
            if float(sum(self.opp_plays))/float(len(self.opp_plays)) > 0.5:
                move = 1
            else:
                move = 0
        self.plays.append(move)
        return move
# c_strength is value from 0-100, probability of rand choosing Coop
class Rand(object):
    # decides randomly between 0 and 1
    def __init__(self, c_strength=75):
        self.name = 'Random'
        self.plays = []
        self.opp_plays = []
        self.round = 0
        self.sum = 0
        # probability of choosing Coop
        self.c_strength = c_strength

    def play(self):
        self.round += 1
        if random() > self.c_strength/float(100):
            move = 1
        else:
            move = 0
        self.plays.append(move)
        return move
def game(p1 = Rand(), p2 = Rand(), rounds=1):
    # takes two heuristics (p1 and p2) and the number of rounds
    # returns the result payoffs from those rounds of the game (payoff_count)

    payoff_count = []
    p1.sum = 0
    p2.sum = 0

    for round in range(rounds):
        d1 = p1.play()
        d2 = p2.play()

        p1.opp_plays.append(d2)
        p2.opp_plays.append(d1)

        payoff = matrix(d1, d2)
        payoff_count.append(payoff)

        p1.sum += payoff[0]
        p2.sum += payoff[1]

def retSum(player=Rand()):
    return player.sum
```

```
def
evolution(numCoop,numDef,numT4T,randPlayers,numMajRep,numRandRep,numEvolution
s, numEachRound):
    totPlayers = []
    totNumPlayers =
numCoop+numDef+numT4T+len(randPlayers)+numMajRep+numRandRep

    randObj = []

    #Put players in list
    if(totNumPlayers%2 !=0):
        sys.exit("Total number of players must be an even number")
    for i in range(numCoop):
        totPlayers.append(Coop())
    for i in range(numDef):
        totPlayers.append(Defect())
    for i in range(numT4T):
        totPlayers.append(T4T())
    for i in range(numMajRep):
        totPlayers.append(MajorityRep())
    for i in range(numRandRep):
        totPlayers.append(RandomRep())

    for num in randPlayers:
        newRand = Rand(num)
        randObj.append(newRand)
        totPlayers.append(newRand)
    countRounds = 0;
    for i in range(numEvolutions):

        countRounds+=1
        coopSum = 0
        defSum = 0
        T4TSum = 0
        majRepSum = 0
        randRepSum = 0
        randomSum = 0

        numRand = len(randObj)

      #Play each matchup
        shuffle(totPlayers)
        it = iter(totPlayers)
        matchups = zip(it,it)
        for matchup in matchups:
            game(matchup[0], matchup[1],numEachRound)

        #Calculate total points for each heuristic
        for player in totPlayers:
            if player.name =="Cooperator":
                coopSum+=player.sum
            elif player.name=="Defector":
                defSum+=player.sum
            elif player.name=="Tit For Tat":
                T4TSum+=player.sum
```

```
            elif player.name =="Majority Reputation":
                majRepSum+=player.sum
            elif player.name =="Random Reputation":
                randRepSum+=player.sum
            else:
                randomSum+=player.sum

        #Add players based on results of last round
        totSum = coopSum+defSum+T4TSum+randomSum+majRepSum+randRepSum

numCoop =int(round(((coopSum)/float(totSum))*totNumPlayers))
        numDef = int(round(((defSum)/float(totSum))*totNumPlayers))
        numT4T = int(round(((T4TSum)/float(totSum))*totNumPlayers))
        numMajRep= int(round(((majRepSum)/float(totSum))*totNumPlayers))
        numRandRep = int(round(((randRepSum)/float(totSum))*totNumPlayers))

        prevNumRand = numRand
        numRand =
int(round(((numRand+randomSum)/float(totSum+totNumPlayers))*totNumPlayers))

        totNumPlayers = numCoop + numDef +
numT4T+numRand+numMajRep+numRandRep
        totPlayers = []

        for i in range(numCoop):
            totPlayers.append(Coop())
        for i in range(numDef):
            totPlayers.append(Defect())
        for i in range(numT4T):
            totPlayers.append(T4T())
        for i in range(numMajRep):
            totPlayers.append(MajorityRep())
        for i in range(numRandRep):
            totPlayers.append(RandomRep())

        #Random:
        randObj.sort(key=retSum,reverse=True)
        newRandObj=[]
        if numRand > prevNumRand:
                for obj in randObj:
                        newObj = Rand(obj.c_strength)
                        totPlayers.append(newObj)
                        newRandObj.append(newObj)

                while prevNumRand<numRand:
                        newObj = Rand(randObj[0].c_strength)
                        totPlayers.append(newObj)
                        newRandObj.append(newObj)
                        prevNumRand+=1
        else:
                count = 0
                for i in range(numRand):
                        newObj = Rand(randObj[count].c_strength)
                        totPlayers.append(newObj)
                        newRandObj.append(newObj)
```

```
                        count+=1
        randObj = newRandObj
#        print(numDef)
 #   return countRounds
       # print(numDef)
     print("Coop: ", numCoop, " Defect: ", numDef, " T4T: ", numT4T, " Rand:
", numRand, " MajRep: ", numMajRep, " RandRep: ", numRandRep, end=' ')
        print(" Random Params: ", end = '')
        for obj in randObj:
            print(obj.c_strength,end= ' ')
        print('')
numArr=[10,10,20,30,40,50,60,70,80,90]
#for i in range(30):
 #       numArr.append(75)

#total = 0.0
#count = 500
#for i in range(count):
#       total+=evolution(100,36,100,numArr,0,0,200,30)

#print(total/count)
evolution(30,30,30,numArr,30,30,50,30)
```