Math 4630/6630 Project Documentation

Using a Genetic Algorithm to Find Optimal Strategies in Iterated Prisoner's Dilemma

Devin Barkey

April 21, 2020

Iterated Prisoner's Dilemma

A classical scenario in game theory, the Prisoner's Dilemma "is an elegant embodiment of the problem of achieving mutual cooperation" [1]. Although many formulations exist, the one-shot two-player variation of the game can be described as follows. Alice and Bob have both been arrested for a crime they allegedly committed together. They are placed in separate interrogation rooms and cannot communicate. Prosecutors give both the opportunity to testify against the other for a suspended sentence. However, this offer comes with a catch. If both "cooperate" with the other and refuse the offer, they will each receive a two year sentence. If one "defects" and testifies against his or her accomplice but the other does not, the defector receives no years in prison while the other receives five years. If both defect, they each receive four years in prison. Thus, Alice and Bob must individually choose whether to cooperate with the other and refuse the offer or defect and testify against the other. This description is summarized in the payoff matrix below.

		Bob	
		Cooperate	Defect
Alice	Cooperate	(3,3)	(0,5)
	Defect	(5,0)	(1,1)

Figure 1: Prisoner's Dilemma payoff matrix where payoffs are 5-[prison sentences]

This results in a "dilemma" as from Alice's and Bob's perspectives, no matter what the other does, his or her best strategy is to defect despite the presence of the mutually beneficial option to both cooperate. The Iterated Prisoner's Dilemma (IPD) extends this setup so that Alice and Bob repeatedly play this game with knowledge of each other's past moves. Given this extension, the question immediately arises of whether there is a strategy which maximizes the average payoff when it is pitted against other strategies.

Genetic Algorithms

Although there is "no rigorous definition of 'genetic algorithm'," most involve "populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring" [3]. The latter three features are termed "operators." The selection operator "selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce." The crossover operator "randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring," and the mutation operator "randomly flips some of the bits in a chromosome" [3]. A genetic algorithm uses these operators to produce successive generations of offspring, which in aggregate should become more fit.

GAs and the IPD

After hosting two computer tournaments for IPD in the late 1970s and early 1980s, political scientist Robert Axelrod endeavored to determine whether the nascent field of genetic algorithms could produce a strategy superior to that which had won the two tournaments. In two different experiments, Axelrod found that his GA outperformed that strategy and produced other intriguing results [2] [3].

In the Python program found below, I implement a variation of Axelrod's second experiment in which strategies compete against one another within the GA [2] [3]. Each strategy is encoded in a 20-character long string of C's and D's. The first 16 characters represent the next move based on the 16 possible results of 2 previous games (CCCC, CCCD, etc.), and the last four represent two hypothetical games used at the start of a round of IPD. The fitness of each strategy is calculated by taking an average of cumulative payoffs from IPD rounds played against other strategies in a generation, including itself. Strategies are then selected using a weighted random sampling to produce offspring. This process is repeated for a given number of generations. The program makes several runs in order to minimize the effect of the inherent randomness in the algorithm.

Thus, this algorithm attempts to solve the following optimization problem:

- variables: 20-characters representing whether to cooperate or defend given the result of two previous games.
- objective function: fitness, i.e. the average score of rounds of IPD played against other strategies in a generation.
- constraints: none other than those inherent in the IPD

References

[1] Robert Axelrod and William D. Hamilton. The evolution of cooperation. *Science*, 211(4489):1390–1396, 1981.

[2] Robert M. Axelrod. *The complexity of cooperation : agent-based models of competition and collaboration / Robert Axelrod.* Princeton studies in complexity. Princeton University Press, Princeton, NJ, 1997.

[3] Melanie Mitchell. *An introduction to genetic algorithms / Melanie Mitchell.* Complex adaptive systems An introduction to genetic algorithms. MIT Press, Cambridge, Mass, 1996.

Effectiveness

In the several tests conducted, the algorithm appeared to be quite sensitive to changes in its parameters. For one set of inputs, the program produced strategies that were indistinguishable in performance from the initial population of random strategies. Another set, such as those used to produce the graphs below, produced better strategies but not always in the last generation. One can infer from this that more experimentation with parameters has the possibility to produce even better results. To do so, however, would require more computational power and, in particular, a parallel implementation of the algorithm to be run on a machine with more cores.

A User's Manual for IPD_GA.py

This user's manual is intended to supplement the in-line comments found in the file itself. As this program was written in Python, one initially needs a Python interpreter or, better yet, a Python IDE to run it.

The Basics

When starting the program, the user will be asked to provide several inputs:

- n: the population size, i.e. the number of strategies in each generation
- g: the number of generations to produce
- runs: the number of runs to perform, where each run consists of g generations

Other inputs have been set by default. However, one can edit the program to ask the user to provide these inputs. They are included below with their default values in parentheses:

- iters: the number of iterations to perform when running the IPD game (150)
- pc: the crossover probability (0.1)
- pm: the mutation probability, i.e. the probability that any given allele in a chromosome string will "flip" (0.001)

Once all inputs have been provided, the program will perform the given number of runs with the given number of generations produced in each. Given the number of steps performed, running information is not provided. Rather, at the end of the program, a plot is produced displaying the mean score of each generation.

An Overview of the Code

Classes

Generator

Class attributes:

- population: a list of Strategy objects which comprise a generation
- size: the population size

Class methods:

- **add**(strat): adds Strategy object strat to Generation's population list.
- **get_pop**(): returns list of chromosomes in population as strings.

Strategy

Class attributes:

- chrom: string (chromosome) of C's and D's which define a strategy

Class methods:

- **move**(mem): returns move defined by chromosome given the results from past two games encoded in mem.
- **get_init_mem**(): returns last four alleles in chromosome defining two hypothetical games used to begin round of IPD.
- **mutate**(pm): performs mutation on chromosome of Strategy object with mutation probability pm. See the section describing genetic algorithms for a further explanation of the mutation operator.

Primary Functions

gen_0(n): creates initial generation of Strategies from randomly generated chromosomes.

- input n: population size.
- returns: Generation object containing randomly generated Strategies.

evolve(cgen, iters, pc, pm): creates next generation.

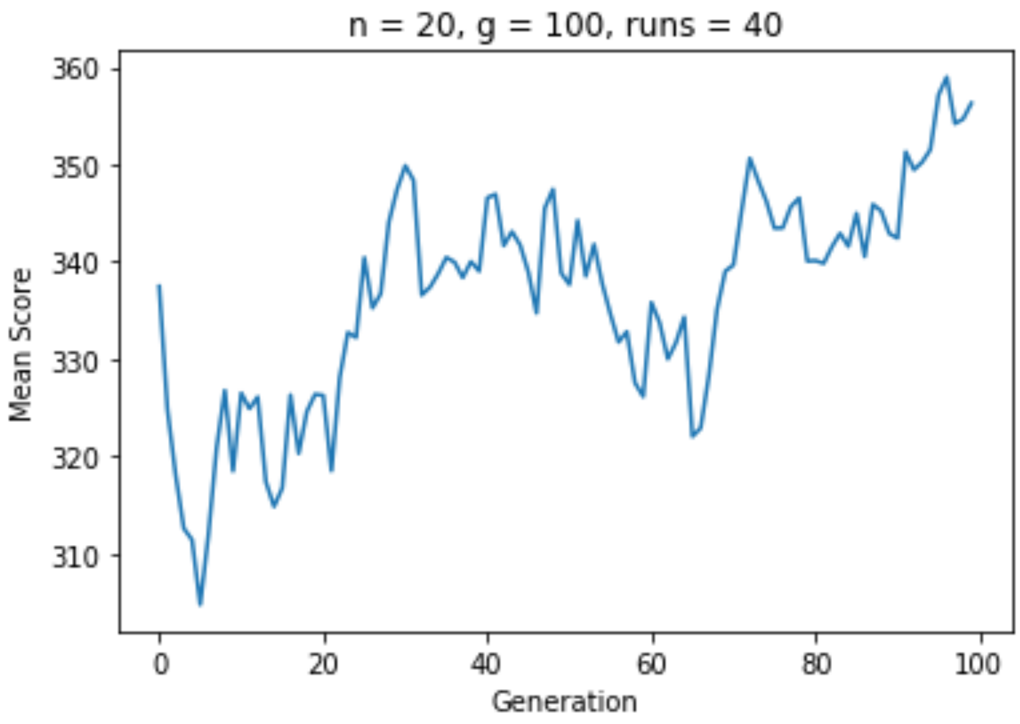
- input cgen: current Generation object.
- input iters: number of iterations to perform in a round of IPD.
- inputs pc, pm: probability of crossover and probability of mutation, respectively.
- returns: next Generation object.

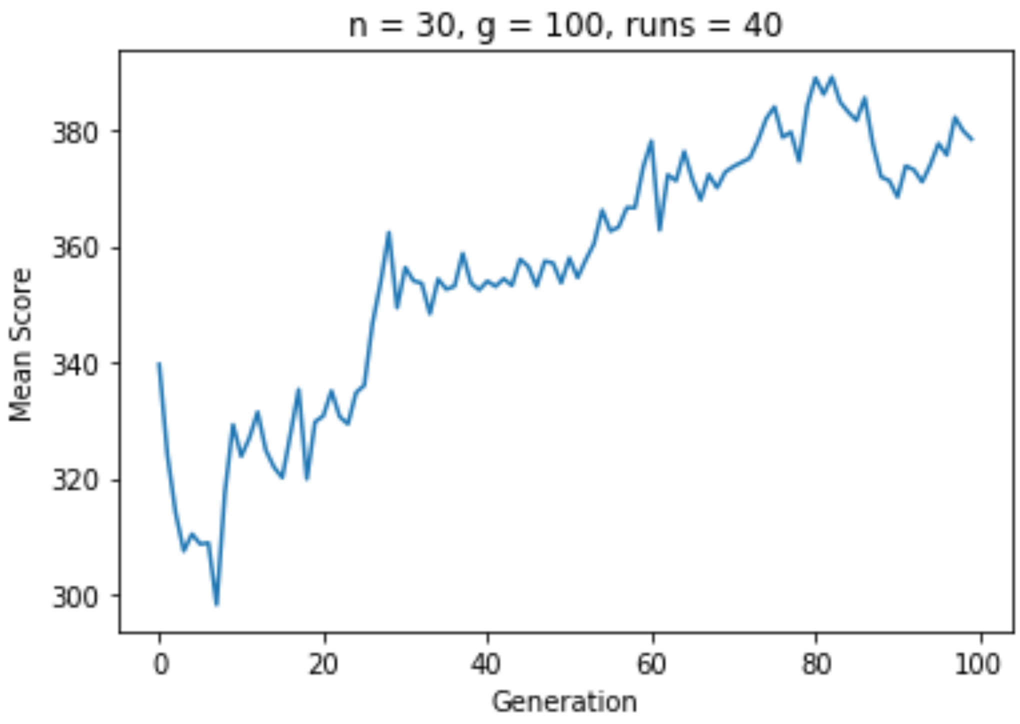
fitness(cgen, iters): calculates fitness scores for each strategy by computing average payoff totals for IPD games against all other strategies in same generation, including itself.

- input cgen: current Generation object.
- input iters: number of iterations to perform in a round of IPD.
- returns: list containing fitness scores for each strategy.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Devin Barkey
Math 4/6630 Project
4/21/20
IPD_GA.py
"""

import random
import numpy as np
import matplotlib.pyplot as plt
from statistics import stdev, mean

"Lookup list containing all possible results of two previous games."
lookup = ["CCCC", "CCCD", "CCDC", "CDCC", "DCCC", "CCDD", "CDCD", "DCCD",
          "CDDC", "DCDC", "DDCC", "CDDD", "DCDD", "DDCD", "DDDC",  "DDDD"]

"Python dictionary in which to look up payoff for player given current moves."
payoffs = {'CC': 3, 'CD': 0, 'DC': 5, 'DD': 1}

##############################################################################
#Classes
##############################################################################

"""
Defines class Generation, which has attributes size and population, the latter
of which contains Strategy objects.
"""
class Generation:

    "Initializes Generation object with empty population list."
    def __init__(self, n):
        self.population = []
        self.size = n

    "Adds Strategy object to Generation's population list."
    def add(self, strat):
        self.population.append(strat)

    "Returns list of chromosomes in population as strings."
    def get_pop(self):
        return [str(strat) for strat in self.population]


"""
Defines class Strategy which has attribute chrom, the
20-character long string of C's and D's that defines a given strategy.
"""
class Strategy:
    "Initializes Strategy object with input chromosome string."
    def __init__(self, chromosome):
        self.chrom = chromosome

    "Returns move defined by chromosome given the results from past two games."
    def move(self, mem):
        ind = lookup.index(mem)
        return self.chrom[ind]

    """
    Returns last four alleles in chromosome defining two hypothetical games
    used to begin round of IPD.
    """
    def get_init_mem(self):
        return self.chrom[16:20]

    "Returns chromosome of Strategy object."
    def __str__(self):
        return self.chrom

    """
    Performs mutation on chromosome of Strategy object by flipping an allele
    with probability pm.
    """
    def mutate(self, pm):
        for i, allele in enumerate(self.chrom):
            if random.random() <= pm:
                if allele == 'C':
                    self.chrom = self.chrom[0:i] + 'D' + self.chrom[i+1:20]
                elif allele == 'D':
                    self.chrom = self.chrom[0:i] + 'C' + self.chrom[i+1:20]

###############################################################################
#Functions
###############################################################################

"Creates initial generation of Strategies from randomly generated chromosomes."
def gen_0(n):
    generation_0 = Generation(n)
    i = 0
    while i < n:
        rchrom = ""

        for j in range(20):
            rchrom += random.choice(["C","D"])

        generation_0.add(Strategy(rchrom))
        i += 1
    return generation_0

"""
```

```
Creates next generation by first computing fitness scores for each strategy.
Then assigns weight of 2 to strategies which perform one standard deviation
above the mean, 1 to strategies within one standard deviation from the mean,
and 0 to strategies below one standard devistion from the mean. Uses these
weights to randomly select parents of offspring. Returns next generation and
mean score of previous generation.
"""
def evolve(cgen, iters, pc, pm):
    nextgen = Generation(cgen.size)
    scores = fitness(cgen, iters)
    mn = mean(scores)
    std = stdev(scores)
    matings = [1 for i in range(cgen.size)]
    for i, score in enumerate(scores):
        if score >= (mn + std):
            matings[i] = 2
        elif score <= (mn - std):
            matings[i] = 0
    j = 0
    while j < cgen.size/2:
        parents = random.choices(cgen.population, weights = matings, k = 2)
        mate(nextgen, parents[0], parents[1], pc, pm)
        j += 1

    return nextgen, mn

"""
Calculates fitness scores for each strategy by computing average payoff totals
for IPD games against all other strategies in same generation, including itself.
"""
def fitness(cgen, iters):
    scores = []

    for player in cgen.population:
        tot_score = 0
        for adversary in cgen.population:
            tot_score += ipd(player, adversary, iters)
        avg_score = tot_score/cgen.size
        scores.append(avg_score)

    return scores

"""
Plays IPD for iters iterations, continuously updating the memory each player
p_1 and p_2.
"""
def ipd(p_1, p_2, iters):
    score = 0
    mem_1 = p_1.get_init_mem()
    mem_2 = p_2.get_init_mem()
    i=0
    while i < iters:
        move = p_1.move(mem_1) + p_2.move(mem_2)
        score += payoffs[move]
        mem_1 = mem_1[2:4] + move
        mem_2 = mem_2[2:4] + move[::-1]
        i += 1

    return score

"""
Produces two offspring from parents par_1 and par_2 with probability of
crossover pc and probability of mutation pm. If no crossover occurs, offspring
are copies of parents. Offspring then added to next generation.
"""
def mate(nextgen, par_1, par_2, pc, pm):
    if random.random() <= pc:
        off_1, off_2 = crossover(par_1, par_2)
    else:
        off_1, off_2 = par_1, par_2
    off_1.mutate(pm)
    off_2.mutate(pm)
    nextgen.add(off_1)
    nextgen.add(off_2)

"""
Performs crossover of two strategy chromosomes x and y by randomly selecting
crossover point and constructing offspring by splicing complementary pieces of
each chromosome together.
"""
def crossover(x, y):
    cp = random.randrange(1,19)
    child_1 = Strategy(str(x)[0:cp] + str(y)[cp:20])
    child_2 = Strategy(str(y)[0:cp] + str(x)[cp:20])
    return child_1, child_2

##############################################################################
#Main
##############################################################################

"""
Main function which prompts user for population size, number of generations,
and number of runs to be performed.
"""
def main():
    n = int(input('Enter the population size n: '))
    g = int(input('Enter the number of generations: '))
    runs = int(input('Enter the number of runs: '))

    iters = 150
    pc = 0.1
    pm = 0.001

    #Optional promts to input desired iters, pc, and pm
    #iters = int(input("Enter number of iterations for PD: "))
    #pc = float(input('Enter the crossover probability Pc: '))
    #pm = float(input('Enter the mutation probability Pm: '))

    "Numpy array to store average scores of each generation in every run."
    record = np.zeros([g, runs])

    j = 0
    while j < runs:
        gen = gen_0(n)

        i = 1
        while i < g:
            gen, pgen_mean = evolve(gen, iters, pc, pm)
            record[i-1,j] = pgen_mean
            i += 1

        record[i-1,j] = mean(fitness(gen, iters))
        j +=1

    "Plot average score for every generation."
    x = np.arange(0,g)
    y = np.mean(record, axis = 1)

    plt.plot(x,y)
    plt.xlabel('Generation')
    plt.ylabel('Mean Score')
    title = 'n = ' + str(n) + ', g = ' + str(g) + ', runs = ' + str(runs)
    plt.title(title)


if __name__ == "__main__":
    main()
```


n = 20, g = 100, runs = 40
Mean Score
360
350
340
330
320
310
Generation
0
20
40
60
80
100


n = 30, g = 100, runs = 40
Mean Score
300
320
340
360
380
0
20
40
60
80
100
Generation