MACQUARIE
University
SYDNEY·AUSTRALIA

MACQUARIE UNIVERSITY
Faculty of Science and Engineering
Department of Computing

COMP3160 Artificial Intelligence 2020 (Semester 2)

Assignment 2 (Report)

Evolutionary Algorithms for Adversarial Game Playing (worth 20%)

Student Name: Liam Phillips

Student Number: 45234817

Student Declaration:

I declare that the work reported here is my own. Any help received, from any person, through discussion or other means, has been acknowledged in the last section of this report.

Student Signature: 

Student Name and Date: Liam Phillips, 29/10/2020

1. Background Knowledge Assessment

a. The Nash equilibrium for the 3PD is P,P,P, this is because for any given actor if you hold the other actors actions constant the score (payoff) will always be higher if the actor chooses P. Therefore it will always end up with all 3 choosing P
Payoff matrix, scores are in order P_i, P_j, P_k

		P_k			
		B		P	
		P_j		P_j	
		B	P	B	P
P_i	B	4,4,4	2,5,2	2,2,5	0,3,3
	P	5,2,2	3,3,0	3,0,3	1,1,1

b. 2 bits for 2 remembered moves, for 3 players. 6 total bits, for 2 bits there are 4 possible arrangements, so there are 4x4x4 = 64 possibilities. Each individual would be 64 bits representing the responses to each possibility + 7 bits for presumes pregame moves. Total 71 bits

c.

It can be just a string of 1's for defects and 0 for cooperation. With this we will have a 71bit long unique binary number as in every different strategy will result in a different number, we can then store the strategy as this number as a decimal or hex. The first bit is the move it will make when there is no history aka round 0. The next 6 bits are the different possibilities for the 2 opponents' first opening moves. Then the 64 bits correspond to each of the 64 possibilities for a 3 player 2 memory size game. With the 2 bits of history combined for all the actors we get a 6 bit long binary number, this corresponds to the index at which the response is selected. The order of the histories when combined doesn't matter as long as it is consistent.

2. Implementation

```
#!/usr/bin/env python

# coding: utf-8


# ### COMP 3160 Assignment 2




import random

import copy



def payoff_to_ind1(individual1, individual2, individual3, game):


    payoff = 0

    x1 = individual1

    x2 = individual2

    x3 = individual3


    if x1 == "1": ##means they selected P

        if x2 != x3:

            return 3

        elif x2 == "1":

            return 1

        else:

            return 5

    else:

        if x2 != x3:

            return 2
```

```
        elif x2 == "1":
            return 0
        else:
            return 4


##strategies are respresented by there 70 responses to the 70 possible histories.
##we can store this as a hex numberto save space
def move_by_ind1(individual1, individual2, individual3, round):

    move = 0 #return var


    #strat bits
    x1 = individual1[2:]
    x2 = individual2[2:]
    x3 = individual3[2:]

    ##get history bits
    h1 = individual1[:2]
    h2 = individual2[:2]
    h3 = individual3[:2]

    ##base cases
    if round < 2:
        if round == 0:
            return x1[0] ##first move
        else:
```

```
        if x1[6] == 1:
            if x2 != x3:
                return x1[5]
            elif x2 == 1:
                return x1[6]
            else:
                return x1[4]
        else:
            if x2 != x3:
                return x1[2]
            elif x2 == 1:
                return x1[3]
            else:
                return x1[1]
else:

    history = h1 + h2 + h3
    his = ''.join(str(i) for i in history)

    index = (int(his, 2)) + 6 ##the pregame moves are the first 6 bits
    ## print(index)
    ##we need to get the bit at index whatever is above

    string = str(x1)
    move = x1[index]

return move ##individual1's move
```

```
#updates the actors memory bits
def process_move(individual, move, memory_depth):
    ind1 = str(individual)
    i = 0
    for i in range(memory_depth -1):
        individual[i+1] = individual[i]
        individual[i] = move
        ind1 = str(individual) + str(ind1[memory_depth:])
    return 1



#this function allows me to specify memory depth and
# n_rounds aswell as get the 2 individuals my actor goes against
#then i pass into the eval_function that gets the actual score
def eval_init(individual):
    score = 0
    mem_depth = 2
    maxSize = len(population) -1
    one = random.randint(0, maxSize)
    two = random.randint(0, maxSize)
    score = eval_function(individual, population[one], population[two], mem_depth, n_rounds)
    return score,



def eval_function(individual1, individual2, individual3, m_depth, n_rounds):
    score = 0
    i = 0
```

```
    ind1 = copy.copy(individual1)
    ind2 = copy.copy(individual2)
    ind3 = copy.copy(individual3)

    for i in range(n_rounds):
        p1 = str(move_by_ind1(ind1,ind2,ind3, i))
        p2 = str(move_by_ind1(ind2,ind3,ind1, i))
        p3 = str(move_by_ind1(ind3,ind1,ind2, i))
        score =  score + payoff_to_ind1(p1,p2,p3, 0)
        process_move(ind1, p1, m_depth)
        process_move(ind2, p2, m_depth)
        process_move(ind3, p3, m_depth)

    return score ##score to individual1



from deap import base
from deap import creator
from deap import tools



def create_toolbox(num_bits):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    # Initialize the toolbox
    toolbox = base.Toolbox()
```

```
    # Generate attributes
    toolbox.register("attr_bool", random.randint, 0, 1)

    # Initialize structures
    toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, num_bits)

    # Define the population to be a list of individuals
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    # Register the evaluation operator
    toolbox.register("evaluate", eval_init)

    # Register the crossover operator
    toolbox.register("mate", tools.cxTwoPoint)

    # Register a mutation operator
    toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)

    # Operator for selecting individuals for breeding
    toolbox.register("select", tools.selTournament, tournsize=3)

    return toolbox


if __name__ == "__main__":
    # Define the number of bits
    num_bits = 73
```

```
#define rounds played
n_rounds = 10

# Create a toolbox using the above parameter
toolbox = create_toolbox(num_bits)

# Seed the random number generator
random.seed(42)

# Create an initial population of 99 individuals
population = toolbox.population(n=99)

# Define probabilities of crossing and mutating
probab_crossing, probab_mutating  = 0.5, 0.05

# Define the number of generations
num_generations = 1000

print('\nStarting the evolution process')
fitnesses = list(map(toolbox.evaluate, population))
for ind, fit in zip(population, fitnesses):
    ind.fitness.values = fit

print('\nEvaluated', len(population), 'individuals')
for g in range(num_generations):
        print("\n===== Generation", g)

        # Select the next generation individuals
```

```
offspring = toolbox.select(population, len(population))

# Clone the selected individuals
offspring = list(map(toolbox.clone, offspring))

# Apply crossover and mutation on the offspring
for child1, child2 in zip(offspring[::2], offspring[1::2]):
    # Cross two individuals
    if random.random() < probab_crossing:
        toolbox.mate(child1, child2)
        # "Forget" the fitness values of the children
        del child1.fitness.values
        del child2.fitness.values

# Apply mutation
for mutant in offspring:
    # Mutate an individual
    if random.random() < probab_mutating:
        toolbox.mutate(mutant)
        del mutant.fitness.values

# Evaluate the individuals with an invalid fitness
invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
fitnesses = map(toolbox.evaluate, invalid_ind)
for ind, fit in zip(invalid_ind, fitnesses):
    ind.fitness.values = fit
```

```
        print('Evaluated', len(invalid_ind), 'individuals')

        # The population is entirely replaced by the offspring
        population[:] = offspring

        # Gather all the fitnesses in one list and print the stats
        fits = [ind.fitness.values[0] for ind in population]

        length = len(population)
        mean = sum(fits) / length
        sum2 = sum(x*x for x in fits)
        std = abs(sum2 / length - mean**2)**0.5

        print("  Min %s" % min(fits))
        print("  Max %s" % max(fits))
        print("  Avg %s" % mean)
        print("  Std %s" % std)

        print("\n==== End of evolution")

best_ind = tools.selBest(population, 1)[0]
print('\nBest individual:\n', best_ind)
```

3. Analysis

a. One thing to note is that the actor with the highest score has a balance between 0's and 1's, this shows us that just responding 1 to all outcomes like expected in the non-iterative prisoner's dilemma. The average score stays similar but could be said to swing back and forth. This is likely the result of certain actors being more greedy and thus getting a higher score, but then in response to that other actors also need to get greedy which brings the average back down to an equilibrium.

b. Tit for tat is a strategy in which the actor responds with whatever its opponent plays. In nIPD this is harder to completely replicate as you are supposed to respond to an opponent with what they play and thus cant completely correctly respond to all opponents unless they all do the same thing. It is interesting to note that in a situation where both opponents do the same thing such as the responses to where they both chose 0 twice in a row is 0 for all 4 cases is also to pick 0 exactly like in tit for tat, which shows us that cooperation does end up coming out on top after enough generations.

5.

4. Notes (Optional)

Wasn't sure which method for representing strategies we should use, even though the 2nd mentioned in the paper is most efficient for player counts that are powers of two in theory it should work for other numbers, so i originally answers 1b,1c using that:

> 1b. 2bits represents the 4 possible last 2 moves of the player recent on the left. 2 other players have 3 possibilities, which is 2bits per move remembered. So 9 possibilities for other actors, multiplied by 4 moves of the player = 36 bits + 6 for the presumed pregame moves is 42
>
> 1c. The individual's previous 2 moves are represented by the first 2 bits on the left then the total defections for the other actors for each move on the right, most recent games are furthest to the left. It would look like:
>
> 00 01 10
>
> “00” being the left is the player's last two moves, so neither move did the player defect. Then 01 is the most recent move of the players, and 1 player defected, finally on the right 10 in binary is 2, so both the 2 other players defected. So the strategy would be the response it takes to each possible combination, which as i calculated in b is 36 possibilities.

Mention here anything worth noting, e.g., whether you faced any particular difficulty in completing any of these tasks, the nature and extent of any help you received from anyone, and why.