MACQUARIE UNIVERSITY
Faculty of Science and Engineering
Department of Computing

COMP3160 Artificial Intelligence 2021 (Semester 2)

Assignment 2 (Report)

Evolutionary Algorithms for Adversarial Game Playing (worth 20%)

Student Name: Joshua Devine

Student Number: 45238278

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: Joshua Devine

Student Name and Date: Joshua Devine 29 /10/2021

1. Background Knowledge Assessment

a) TC1 as the more each cryptocurrency miner or holder chooses to pursue increasing their portfolio it will eventually come at the cost of the environment. Specifically natural resources needed to produce energy i.e. coal will runout, climate change will be accelerated thus endangering the world.

b)

Dominant Strategy

		Row	
		C	**D**
Column	**C**	**(5,5)**	$\underline{\textbf{(4,8)}}$
	D	$\underline{\textbf{(8,4)}}$	**(0,0)**

- For column it is to play D as 8 > 5 and for row to play C as 4 > 0
- For row it is to play D as 8 > 5 and for column is to play C as 4 > 0

Dominant Strategy Equilibrium

(**D**,**C**) or (**D**,**C**)

Nash Equilibrium

		Row	
		C	**D**
Column	**C**	**(5,5)**	$\underline{\textbf{(4,8)}}$
	D	$\underline{\textbf{(8,4)}}$	**(0,0)**

For row playing C, columns best move is D as they will get their highest unit's payoff of 8. Whereas column playing C, rows best move is D as they will get their highest payoff of 8. Thus, the moves are (**C**,**D**) and (**C**,**D**).

Pareto Optimal Strategy

(**C**,**D**) and (**C**,**D**) if row played C column would deviate to D as that's where they would get their highest payoff but it would come at a cost for row as they would only get a payoff of 4. If column played C row would deviate to D as that's where they would get their highest cost of 8 but it would come at a cost for column as they would only get a payoff of 4.

c)

Dominant Strategy

		Row	
		C	D
Column	C	**(12,12)**	**(11,13)**
	D	**(13,11)**	**(12,12)**

- For column playing D would be their best choice as 13 > 12, 12 > 11 and row would also play D as it is their best choice 13 > 12, 12 > 11

Dominant Strategy Equilibrium

(**D**,**D**)

Nash Equilibrium

		Row	
		C	D
Column	C	**(12,12)**	**(11,13)**
	D	**(13,11)**	**(12,12)**

Row playing C, columns best move is D but that would result in not the best move for row C as 11 < 12. Column playing D, rows best move would be D as it results in the highest unit's payoff for both column and row. Thus, the move is (**D**,**D**).

Pareto Optimal Strategy

(**C**,**D**) and (**C**,**D**) if row played C column would deviate to D as that's where they would get their highest payoff but it would come at a cost for row as they would only get a payoff of 11. If column played C row would deviate to D as that's where they would get their highest cost of 13 but it would come at a cost for column as they would only get a payoff of 11.

d) Whilst running the plays for 1(b) and 1(c) the dominant strategy and nash equilibrium were the same results for both however 1(b) dominant strategy had 2 possible selections whereas 1(c) had 1. This suggested that the dominant strategy showed how these plays were dominant whereas the nash equilibrium explained why.

e) If the tit for tat strategy was played then for both ITC(1) and ITC(2) after the first move of cooperating the strategy would nominate moves based on the opponents move. Thus, depending on the player there would either be cooperation or defecting but no mixed moves.

2. Implementation

a)

```
def payoff_to_player1(player1, player2, game):
    # your code goes here
    tc_matrix_val = str(player1[len(player1)-1]) + str(player2[len(player2)-
1])
    print(game)
    payoff = game.get(tc_matrix_val)
    return payoff
```

b)

```
def next_move(player1, player2, round):
    #your code goes here
    tmp = 0
    decimal_val = 0
    player1_move = []
    pl1_pos = 0
    pl2_pos = 0
    if round == 0 or (len(player1) == 18 and len(player2) == 18): #checks if
round is 0 meaning the game has just started or if either of players lengths
are 18 meaning they have no memory bits
        for idx in range(len(player1)):
            if (len(player1) == 18 and len(player2) == 18) and (idx == 16 or
idx == 17): #if the length of both players is 18 then the default bits will be
appended to the end of each player and used as a move
                player1.append(player1[idx])
                player2.append(player2[idx])
            if idx == 17:
                player1_move.append(player1[idx])
                player1_move.append(player2[idx])
        return player1_move
    elif round >= 1:
        tmp = str(player1[len(player1)-1]) + str(player1[len(player1)-2]) +
str(player2[len(player2)-1]) + str(player2[len(player2)-2])
        decimal_val = int(tmp, 2)
        pl1_pos = player1[decimal_val]
        pl2_pos = player2[decimal_val]
        player1_move.append(pl1_pos)
        player1_move.append(pl2_pos)
    return player1_move #returns list with moves for player1 and player2 found
in their respective strategy bits
```

c)

```
def process_move(player, move, memory_depth):
    # your code goes here
    del player[len(player)-memory_depth]#deletes old memory bits and adds on
new bits retrieved from the strategy bits
    player.append(move)
    if len(player) == 20 and player[len(player)-1] == move: #ensures if the
process was done correctly by checking the size of the player and new memory
bits values are the same as the move
        return True
    else:
        return False
```

d)

```
def score(player1, player2, m_depth, n_rounds, game):
    # your code goes here
    score_to_player1 = 0
    score_to_player2 = 0
    move_list = []
    move_pl1 = 0
    move_pl2 = 0
    payoff_pl = 0
    #print("pl1" , player1)
    #print("pl2" , player2)
    for i in range(n_rounds):
        move_list = []
        move_list = next_move(player1, player2, i)
        move_pl1 = move_list[0]
        move_pl2 = move_list[1]
        process_move(player1, move_pl1, m_depth) #process both player1 and
player2 to ensure they are both up to date with current data and reflects the
correct functionality of the spec
        process_move(player2, move_pl2, m_depth)
        payoff_pl = payoff_to_player1(player1, player2, game)
        score_to_player1 += payoff_pl[0]
        score_to_player2 += payoff_pl[1]
    return score_to_player1
```

e) i) and ii)

```
def eval_func(individual, individual2):
    game = tc
    return score(individual,individual2,mem_depth,n_rounds,game),

# Create the toolbox with the right parameters
def create_toolbox(num_bits):
    # your code goes here

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("individual", list, fitness=creator.FitnessMax)


    toolbox = base.Toolbox()
    toolbox.register("attr_bool", random.randint, 0, 1)
    toolbox.register("individual", tools.initRepeat, creator.individual,
                 toolbox.attr_bool, num_bits)
    toolbox.register("population1", tools.initRepeat, list,
toolbox.individual)
    toolbox.register("population2", tools.initRepeat, list,
toolbox.individual)
    toolbox.register("evaluate", eval_func)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
    toolbox.register("select", tools.selTournament, tournsize=3)
    return toolbox


# This function implements the evolutionary algorithm for the game
def play_game(mem_depth, population1_size, generation_size, n_rounds, game):
    mem_depth = 2
    num_bits = pow(2,(2*mem_depth))+(2*mem_depth)   # your code goes here:
calculate the bits using the mem_depth value

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

    # Seed the random number generator
    random.seed(3)

    # Create an initial population1 of n individuals
    population1 = toolbox.population1(n = population1_size)
    population2 = toolbox.population2(n = population1_size)

    # Define probabilities of crossing and mutating
    probab_crossing, probab_mutating  = 0.5, 0.2

    print('\nStarting the evolution process')

    # Evaluate the entire population1
    # your code goes here:

    #Using DEAP documentation and week 8 tutorial the onemax GA was applied
    fitnesses = list(map(toolbox.evaluate, population1, population2))

    # Calculate the fitness value for each player.
    # Each player will play against every other player in the population1.
    # The fitness values of a player is the total score of all games played
against every other players.

    print('\nEvaluated', len(population1), 'individuals')

    for ind, fit in zip(population1, fitnesses):
        ind.fitness.values = fit

    # Iterate through generations
    for g in range(generation_size):
        print("\n===== Generation", g)
        # your code goes here
        # apply the steps of the evolutionary algorithm

          # Extracting all the fitnesses of
        fits = [ind.fitness.values[0] for ind in population1]
        offspring = toolbox.select(population1, len(population1))
        offspring = list(map(toolbox.clone, offspring))
        for child1, child2 in zip(offspring[::2], offspring[1::2]):
            if random.random() < probab_crossing:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values
```

```python
        for mutant in offspring:
            if random.random() < probab_mutating:
                toolbox.mutate(mutant)
                del mutant.fitness.values
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit
        print("  Evaluated %i individuals" % len(invalid_ind))
        population1[:] = offspring
        fits = [ind.fitness.values[0] for ind in population1]
        length = len(population1)
        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("-- End of (successful) evolution --")


if __name__ == "__main__":
    mem_depth = 2
    population1_size = 10
    generation_size = 5
    n_rounds = 4
    tc = 0

    print('===================')
    print('Play the game ITC1')
    print('===================')
    tc = tc1_payoffs
    play_game(mem_depth, population1_size, generation_size, n_rounds, tc)

    print('\n\n===================')
    print('Play the game ITC2')
    print('===================')
    tc = tc2_payoffs
    play_game(mem_depth, population1_size, generation_size, n_rounds, tc)
```

3. Analysis

a) The best genetic algorithm generated was the onemax algorithm using a fitness max function with a weight of 1.0, a probability of crossing over = 0.5 and a mutation rate of indpb = 0.05.

b) No, it did not align with my prediction where it was believed the players next move would be same as their next moves would be determined on what one player chose. Rather it seems the players are playing to get their best possible payoff.

c) The best genetic algorithm generated was the onemax algorithm using a fitness max function with a weight of 1.0, a probability of crossing over = 0.5 and a mutation rate of indpb = 0.05.

d) No, it did not align with my prediction where it was believed the players next move would be same as their next moves would be determined on what one player chose. Rather it seems the players are playing to get their best possible payoff.

e) Any industry which relies on a finite number of resources like the fishing industry has correlations between the results generated. Since the intention of each business in this industry is to maximize sales, they achieve this by selling more fish. They may gain a higher payoff in certain scenarios however as seen in the results there are cases where the payoff is less or none which is a result of overdemand on the supplies of fish to the point of exhausted supply.

4. Notes (Optional)

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.

The lack of instruction and obscureness constantly reoccurring throughout the spec document for this assessment made it a lot harder than it needed to be.