MACQUARIE
University
SYDNEY·AUSTRALIA

**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: Runde Jia**

**Student Number: 44434065**


**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.*

贾润德

Student Signature:

Student Name and Date: Runde Jia 10/29/2021

1. Background Knowledge Assessment
   a) TC1 is closer. Because in bitcoin mining, everyone wants to invest more computing power get more bitcoin, but this makes environment worse, and may not continue bitcoin mining for everyone. There is no contract between two players.

   b) Game TC1:
   Dominant Strategy Equilibria:

   No dominant strategy for column
   No dominant strategy for row
   So, there is no dominant strategy equilibrium.

   Nash Equilibria:
   (D,C) or (C,D)

   Pareto Optimal: (C,C), (C,D),(D,C)

   c) Game TC2:

   Dominant Strategy Equilibria:

   A dominant strategy for column is playing D
   A dominant strategy for row is playing D
   So (D, D) is a dominant strategy equilibrium

   Nash Equilibria:
   (D,D)

   Pareto Optimal: (C,D) or (D,C)

   d) TC1 has two Nash Equilibria (D, C) and (C, D), players would receive different results. And in these two Nash Equilibria, two players need to coordinate to maximize their benefits.

   In TC2, they only have one Nash Equilibria, two players can keep choose (D,D) to receive best results.

e) Iterated TC1: Two players would choose C or D in certain ratio, cooperate in some situations to get best results.

Iterated TC2: Both players would choose D, because it is the only one Nash Equilibria.

## 2. Implementation

a)
```
def payoff_to_player1(player1, player2, game):
payoff = game[player1][player2]
   return payoff
```

b)
```
def next_move(player1, player2,round):
    m_depth = 2
    strat_bit = 2**(2*m_depth)
   player1_move = player1[strat_bit + round] if round < m_depth else
player1[int('0b' + str(player1[-2]) + str(player2[-2]) + str(player1[-1]) +
str(player2[-1]),2)]

   return player1_move
```

c)
```
def process_move(player, move, m_depth):
    memory_strat_bit = 2**(2*m_depth) + m_depth
   player[memory_strat_bit],player[memory_strat_bit + 1] =
player[memory_strat_bit + 1],move
```

d)
```
def score(player1, player2, m_depth, n_rounds, game):
    score_to_player1  = 0
    for round in range(n_rounds):
        p1_move = next_move(player1,player2,round)
        p2_move = next_move(player2,player1,round)
        c_score = payoff_to_player1(p1_move, p2_move, game)
        process_move(player1,p1_move,m_depth)
        process_move(player2,p2_move,m_depth)
```

```
                score_to_player1 += c_score

            return score_to_player1
```

i.

```
def create_toolbox(num_bits):

    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,
n = num_bits)

    toolbox.register('population', tools.initRepeat, list, toolbox.individual)

    toolbox.register('selTournament', tools.selTournament, tournsize = 2)

    toolbox.register("evaluate", score)

    return toolbox




# This function implements the evolutionary algorithm for the game

def play_game(mem_depth, population_size, generation_size, n_rounds,
game,crossing,mutating):

    mem_depth = 2

    num_bits = 2**(2*mem_depth) + 2*mem_depth
```

```
# Create a toolbox using the above parameter

toolbox = create_toolbox(num_bits)


# Seed the random number generator

random.seed(3)


# Create an initial population of n individuals

population = toolbox.population(n = population_size)


# Define probabilities of crossing and mutating

probab_crossing, probab_mutating  = crossing,mutating


print('\nStarting the evolution process')


# Evaluate the entire population

fitnesses = []

    # your code goes here:

    # Calculate the fitness value for each player.

    # Each player will play against every other player in the population.

    # The fitness values of a player is the total score of all games played against
every other players.
```

```
    for i in population:
        f_scores = sum([score(i,other, mem_depth, n_rounds, game) for other in population])
        fitnesses.append((f_scores,))
        i.fitness.values = (f_scores,)

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

    # Iterate through generations
    for g in range(generation_size):
        print("\n===== Generation", g)

        Tour = toolbox.selTournament(population,3)

        # crossing use cxTwoPoint
        gn1,gn2,gn3 = [toolbox.clone(ind) for ind in Tour]
        crossing = [gn1,gn2,gn3]
        if random.random() < probab_crossing:
            crossing1,crossing2 = random.sample([gn1,gn2,gn3],2)
            tools.cxTwoPoint(crossing1,crossing2)
            crossing.extend([crossing1,crossing2])
```

```
# mutant use mutFlipBit
generation = []
for cross in crossing:
    if random.random() < probab_mutating:
        mutant = toolbox.clone(cross)
        tools.mutFlipBit(mutant, 0.05)
        generation.append(deepcopy(mutant))
    else:
        generation.append(deepcopy(cross))


# add new generations
population.extend(generation)
# calculate the fitness values


fitnesses = []
for i in population:
    f_scores = sum([score(i,other, mem_depth, n_rounds, game) for other in population])
    fitnesses.append((f_scores,))
    i.fitness.values = (f_scores,)
# select the best
population = tools.selBest(population,population_size)
```

```
    for individual in population:
        print("The fitness value: {} the strategy:
{}".format(individual.fitness.values,individual))



if __name__ == "__main__":
    mem_depth = 2
    population_size = 10
    generation_size = 5
    n_rounds = 4

    print('==================')
    print('Play the game ITC1')
    print('==================')
    play_game(mem_depth, population_size, generation_size, n_rounds,
tc1_payoffs,0.5,0)
```

ii.

```
def create_toolbox(num_bits):
    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, n = num_bits)
```

```
        toolbox.register('population', tools.initRepeat, list,
toolbox.individual)
        toolbox.register('selTournament', tools.selTournament,
tournsize = 2)
        toolbox.register("evaluate", score)
        return toolbox



    # This function implements the evolutionary algorithm for the
game
    def play_game(mem_depth, population_size, generation_size,
n_rounds, game,crossing,mutating):
        mem_depth = 2
        num_bits = 2**(2*mem_depth) + 2*mem_depth

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

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

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

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

        print('\nStarting the evolution process')

        # Evaluate the entire population
        fitnesses = []
          # your code goes here:
          # Calculate the fitness value for each player.
          # Each player will play against every other player in the
population.
```

```
        # The fitness values of a player is the total score of all
games played against every other players.
      for i in population:
         f_scores = sum([score(i,other, mem_depth, n_rounds,
game) for other in population])
         fitnesses.append((f_scores,))
         i.fitness.values = (f_scores,)

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

      # Iterate through generations
      for g in range(generation_size):
         print("\n===== Generation", g)

         Tour = toolbox.selTournament(population,3)

         # crossing use cxTwoPoint
         gn1,gn2,gn3 = [toolbox.clone(ind) for ind in Tour]
         crossing = [gn1,gn2,gn3]
         if random.random() < probab_crossing:
            crossing1,crossing2 =
random.sample([gn1,gn2,gn3],2)
            tools.cxTwoPoint(crossing1,crossing2)
            crossing.extend([crossing1,crossing2])

         # mutant use mutFlipBit
         generation = []
         for cross in crossing:
            if random.random() < probab_mutating:
               mutant = toolbox.clone(cross)
               tools.mutFlipBit(mutant, 0.05)
               generation.append(deepcopy(mutant))
            else:
               generation.append(deepcopy(cross))

         # add new generations
```

```
        population.extend(generation)
        # calculate the fitness values

        fitnesses = []
        for i in population:
            f_scores = sum([score(i,other, mem_depth, n_rounds,
game) for other in population])
            fitnesses.append((f_scores,))
            i.fitness.values = (f_scores,)
        # select the best
        population = tools.selBest(population,population_size)

    for individual in population:
        print("The fitness value: {} the strategy:
{}".format(individual.fitness.values,individual))


if __name__ == "__main__":
    mem_depth = 2
    population_size = 10
    generation_size = 5
    n_rounds = 4

    print('\n\n===================')
    print('Play the game ITC2')
    print('===================')
    play_game(mem_depth, population_size, generation_size,
n_rounds, tc2_payoffs,0.9,0.9)
```

## 3. Analysis

a) The maximum fitness value is 330.0, the best strategy is [1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]
   Type of Crossover: cxTwoPoint
   Type of Mutation: mutFlipBit (Independent probability = 0.05)
   Probab_crossing = 0.5
   Probab_mutating = 0

b) The strategy chooses D most of the time to improve the overall income, sometimes chooses C. It does not align with the prediction in 1e.

c) The maximum fitness value is 745.0, the best strategy is [0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1]
   Type of Crossover: cxTwoPoint
   Type of Mutation: mutFlipBit(Independent probability = 0.05)
   Probab_crossing = 0.9
   Probab_mutating = 0.9

d) The strategy chooses D most of the time, but sometimes it chooses C that violates the prediction. This variation makes better outcome for the players

e) All bitcoin mining companies should cooperate, no overexploitation. They should make a deal, tacit understanding. Protect the environment, to make sure there is max benefit for each one.

## 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.*

Finding probab_crossing and probab_mutating.