AI lab 4

part b

Rock paper scissors

Zinat Abo Hgool 206721714

Elie haddad 207931536

Q1-4

Our bots

Anti Flat: deterministic/ predicts opponent's move/exploits

Copy: deterministic because it always plays the winning move against the opponents last move/ meta strategic/ exploits

Freq: plays the winning move against the most frequent opponent's move /deterministic/combination of strategies it tries to predict the opponent move based on past moves

Flat: stochastic/random since it uses coin flips/ partially exploits and explores

Foxtrot: stochastic/ random since it uses a coin flip/ partially explorative

Bruijn81: deterministic it uses constant values so it's not random and doesn't use meta strategies/

Pi: deterministic since the value of pi is constant /partially exploits and explores

Play226: stochastic/use random moves based on preset probabilities/no meta strategies

RndPlayer: stochastic/ plays random moves /doesn't learn or predict since it just plays randomly

Rotate: deterministic because it follows a pattern rock-> paper-> scissors /doesn't learn nor predict/ doesn't use a meta strategy

Switch: stochastic /partially random/tries to learn from the opponent last move/exploits

SwitchAlot: stochastic since it uses random values it's like switch but it might pick the previous choice sometimes/ exploits the player's last move

Q5-8

Our player

We programmed a rock paper scissors player to compete with the other bots and hopefully win

We used coevolution and the genetic algorithm concept to build it

First we initialized our population which consists of arrays with 1000 randomly generated moves then we initiated the parasite population which is a group of bots

```python
def init_population(self):  # create popsize citizens

    for i in range(self.args.GA_POPSIZE):  # initialize the agents population
        array = [random.randint(0, 2) for i in range(1000)]
        player = Agent(array, 0)
        self.population.append(player)

    for j in range(1):
        self.parasites.append(Dummy.AntiFlat())
        self.parasites.append(Dummy.Copy())
        self.parasites.append(Dummy.Freq())
        self.parasites.append(Dummy.Flat())
        self.parasites.append(Dummy.Foxtrot())
        self.parasites.append(Dummy.Bruijn81())
        self.parasites.append(Dummy.Pi())
        self.parasites.append(Dummy.Play226())
        self.parasites.append(Dummy.RndPlayer())
        self.parasites.append(Dummy.Rotate())
        self.parasites.append(Dummy.Switch())
        self.parasites.append(Dummy.SwitchALot())

def calc_fitness(self, population: list[Agent]):
```

Then to calculate the fitness of each player in the population we matched it against every bot in the parasite and the fitness was the number of losses it had against them

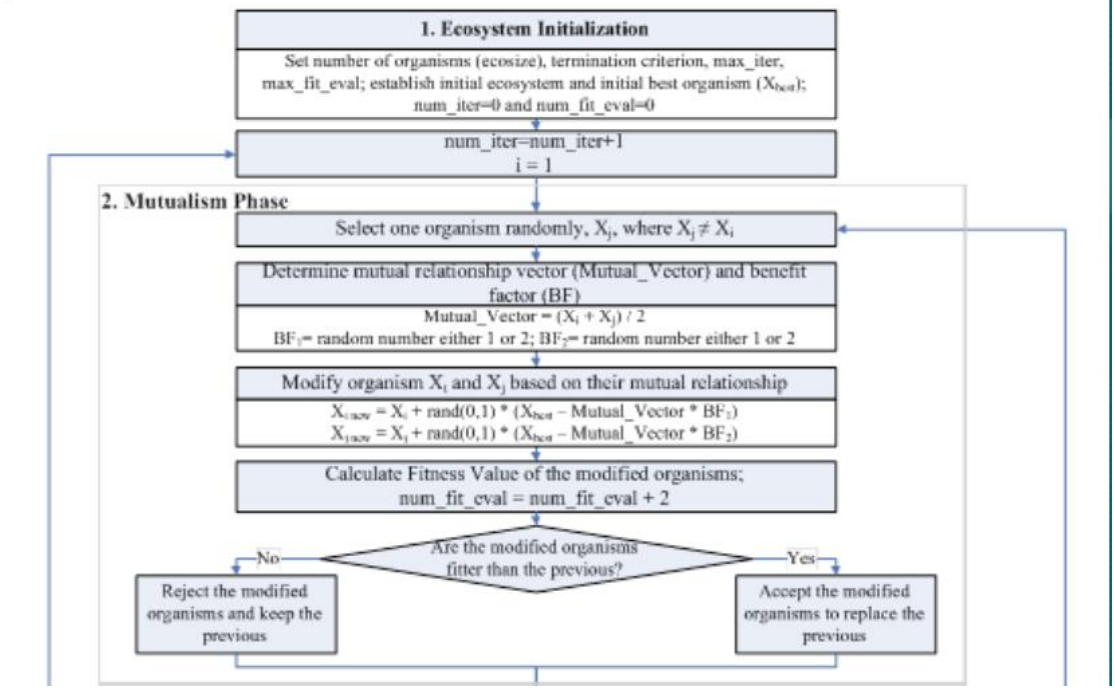
We build a function called Result to check who won each round and if by the end of all rounds the result was >0 then we won

Since win=1,draw=0,loss=-1

```python
def calc_fitness(self, population: list[Agent]):
    loss = 0
    rounds = 0
    for pop in self.population:
        for par in self.parasites:
            par.newGame(len(pop.arr))
            for i in range(len(pop.arr)):
                r = self.result(pop.arr[i], par.nextMove())
                rounds += r
                par.storeMove(pop.arr[i], r)
            if rounds < 0:
                loss += 1
            rounds = 0
        pop.fitness = loss
        loss = 0

def result(self, p1, p2):
    if ((p2 == 0 and p1 == 1) or (p2 == 1 and p1 == 2) or (p2 == 2 and p1 == 0)):
        return 1
    if ((p2 == 0 and p1 == 0) or (p2 == 1 and p1 == 1) or (p2 == 2 and p1 == 2)):
        return 0
    return -1
```

And then we need to construct our new generation and so we used the formulas we learned in the lecture to implement the three functions Mutualism, Communalism, Parasitism

It’s important to say that we saved the best global player and after each generation we compared it with the current generation’s best player and if it was better we would switch them


1. Ecosystem Initialization
Set number of organisms (ecosize), termination criterion, max_iter, max_fit_eval; establish initial ecosystem and initial best organism (X_{best}); num_iter=0 and num_fit_eval=0
num_iter=num_iter+1
i = 1
2. Mutualism Phase
Select one organism randomly, X_j, where $X_j \neq X_i$
Determine mutual relationship vector (Mutual_Vector) and benefit factor (BF)
Mutual_Vector = (X_i + X_j) / 2
BF_1= random number either 1 or 2; BF_2= random number either 1 or 2
Modify organism X_i and X_j based on their mutual relationship
$X_{inew} = X_i + rand(0,1) * (X_{best} - Mutual\_Vector * BF_1)$
$X_{jnew} = X_j + rand(0,1) * (X_{best} - Mutual\_Vector * BF_2)$
Calculate Fitness Value of the modified organisms; num_fit_eval = num_fit_eval + 2
Are the modified organisms fitter than the previous?
No
Yes
Reject the modified organisms and keep the previous
Accept the modified organisms to replace the previous

Mutualism

```
def mate(self):
    self.elitism(self.population, self.buffer)
    for i in range(self.esize, self.args.GA_POPSIZE):
        i1 = i
        # [num1 * num2 for num1, num2 in zip(a, b)]
        while i1 == i:
            i1 = randint(0, (self.args.GA_POPSIZE) - 1)
        bf1 = randint(1, 2)
        bf2 = randint(1, 2)
        # mv=np.add(self.population[i].arr,self.population[i1].arr)
        mv = [num1 + num2 for num1, num2 in zip(self.population[i].arr, self.population[i1].arr)]
        # mv=np.divide(mv,2)
        res = [int(num1 / 2) for num1 in mv]
        mv = res
        mv2 = mv
        res = [num * bf1 for num in mv]
        res2 = [abs(num1 - num2) for num1, num2 in zip(self.best, res)]
        res3 = [int(num * uniform(0, 1)) for num in res2]
        gene = [num + num2 for num, num2 in zip(self.population[i].arr, res3)]
        mv = mv2
        res = [num * bf2 for num in mv]
        res2 = [abs(num1 - num2) for num1, num2 in zip(self.best, res)]
        res3 = [int(num * uniform(0, 1)) for num in res2]
        gene2 = [num + num2 for num, num2 in zip(self.population[i1].arr, res3)]

        # gene=self.population[i].arr+np.multiply(randint(0,1),(np.subtract(self.best,np.multiply(mv
        # print(gene)
```

Communalism

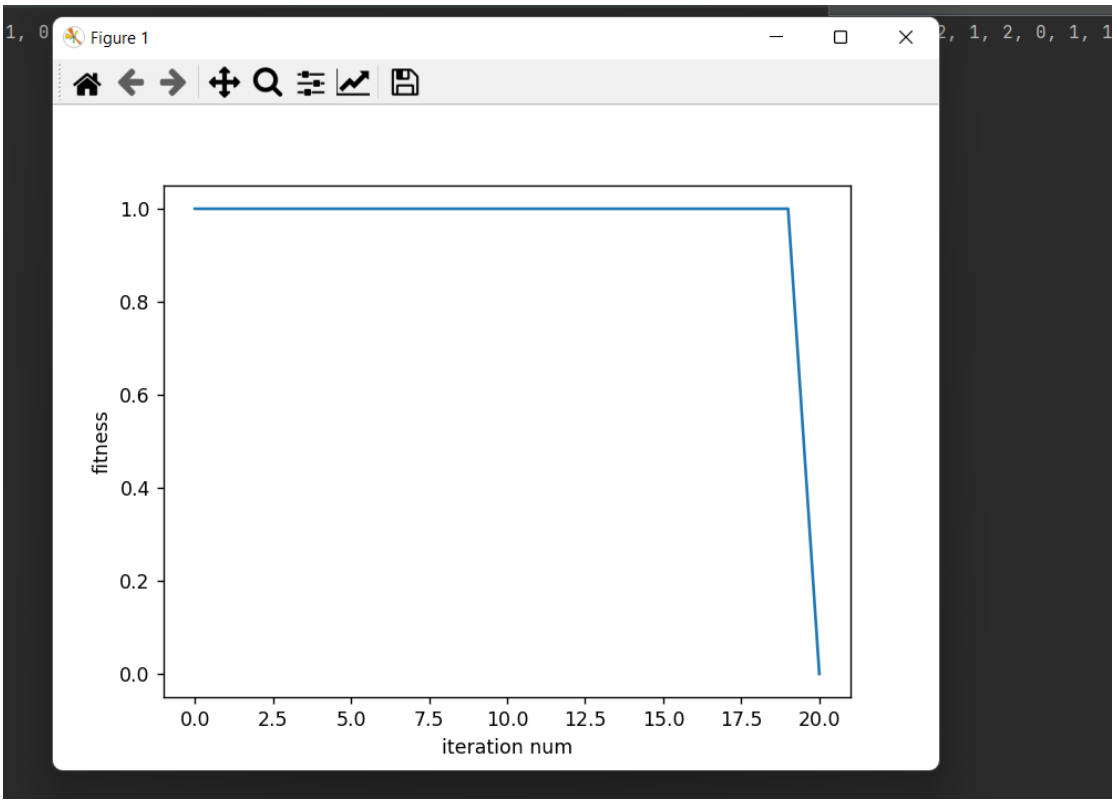
```
def commensalism(self):
    for i in range(self.esize, self.args.GA_POPSIZE):
        i1 = i
        while i1 == i:
            i1 = randint(0, (self.args.GA_POPSIZE) - 1)

        res2 = [abs(num1 - num2) for num1, num2 in zip(self.best, self.population[i].arr)]
        res3 = [num * random.choice([-1, 1]) for num in res2]
        gene = [num + num2 for num, num2 in zip(self.population[i].arr, res3)]

        # gene=self.population[i].arr+randint(-1,1)*np.subtract(self.best,self.population[i].arr)
        loss = 0
        rounds = 0

        for par in self.parasites:
            par.newGame(len(gene))
            for k in range(len(gene)):
                r = self.result(gene[k], par.nextMove())
                rounds += r
                par.storeMove(gene[k], r)
            if rounds < 0:
                loss += 1
            rounds = 0

        if (loss < self.population[i].fitness):
            self.population[i] = Agent(gene, loss)
```

Parasitism

```
def paraisitsm(self):
    for i in range(self.esize, self.args.GA_POPSIZE):
        i1 = i
        while i1 == i:
            i1 = randint(0, (self.args.GA_POPSIZE) - 1)
        pv = self.population[i].arr
        for j in range(len(self.population[i].arr)):
            if uniform(0, 1) < self.args.GA_MUTATION:
                pv[j] = randint(0, 2)
        loss = 0
        rounds = 0

        for par in self.parasites:
            par.newGame(len(pv))
            for k in range(len(pv)):
                r = self.result(pv[k], par.nextMove())
                rounds += r
                par.storeMove(pv[k], r)

            if rounds < 0:
                loss += 1

            rounds = 0

        if (loss < self.population[i].fitness):
            self.population[i] = Agent(pv, loss)
```

And by the end we managed to get to our player that beat all of the other bots

```
fitness:  1
best player:  [1, 0, 2, 0, 2, 1, 1, 1, 2, 0, 0, 1, 1, 2, 2, 1, 1, 2, 0, 2, 2, 2, 2, 0, 2, 1, 1, 0, 2, 0, 1, 2, 2, 1, 0, 2, 1, 1, 0, 2, 2, 1, 2, 0, 2, 2, 0, 2, 2, 2, 0, 2, 1, 2, 0,
Clock ticks time:  7.236291170120239
fitness:  1
best player:  [1, 0, 2, 0, 2, 1, 1, 1, 2, 0, 0, 1, 1, 2, 2, 1, 1, 2, 0, 2, 2, 2, 2, 0, 2, 1, 1, 0, 2, 0, 1, 2, 2, 1, 0, 2, 1, 1, 0, 2, 2, 1, 2, 0, 2, 2, 0, 2, 2, 2, 0, 2, 1, 2, 0,
Clock ticks time:  7.241595029830933
fitness:  1
best player:  [1, 0, 2, 0, 2, 1, 1, 1, 2, 0, 0, 1, 1, 2, 2, 1, 1, 2, 0, 2, 2, 2, 2, 0, 2, 1, 1, 0, 2, 0, 1, 2, 2, 1, 0, 2, 1, 1, 0, 2, 2, 1, 2, 0, 2, 2, 0, 2, 2, 2, 0, 2, 1, 2, 0,
Clock ticks time:  7.259379148483276
fitness:  1
best player:  [1, 0, 2, 0, 2, 1, 1, 1, 2, 0, 0, 1, 1, 2, 2, 1, 1, 2, 0, 2, 2, 2, 2, 0, 2, 1, 1, 0, 2, 0, 1, 2, 2, 1, 0, 2, 1, 1, 0, 2, 2, 1, 2, 0, 2, 2, 0, 2, 2, 2, 0, 2, 1, 2, 0,
Clock ticks time:  7.251201629638672
fitness:  1
best player:  [1, 0, 2, 0, 2, 1, 1, 1, 2, 0, 0, 1, 1, 2, 2, 1, 1, 2, 0, 2, 2, 2, 2, 0, 2, 1, 1, 0, 2, 0, 1, 2, 2, 1, 0, 2, 1, 1, 0, 2, 2, 1, 2, 0, 2, 2, 0, 2, 2, 2, 0, 2, 1, 2, 0,
Clock ticks time:  7.438626289367676
fitness:  0
best player:  [1, 0, 0, 1, 0, 0, 0, 0, 2, 1, 0, 1, 2, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 2, 0, 0, 2, 2, 1, 2, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 2, 0, 2, 1, 1, 1, 1, 1, 0, 1,

Best string:  [1, 0, 0, 1, 0, 0, 0, 0, 2, 1, 0, 1, 2, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 2, 0, 0, 2, 2, 1, 2, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 2, 0, 2, 1, 1, 1, 1, 1, 0, 1,
found in 20 iterations out of 100
Elapsed time:  148.1357796
average against  Anti Flat Player  is:  3.6
sd against  Anti Flat Player is:  0.8
average against  Copy Player  is:  77.36
sd against  Copy Player is:  73.76867627929892
average against  Freq Player  is:  18.736
sd against  Freq Player is:  75.30973440399322
average against  Flat Player  is:  4.2736
sd against  Flat Player is:  75.3169956448078
average against  Foxtrot Player  is:  5.52736
```



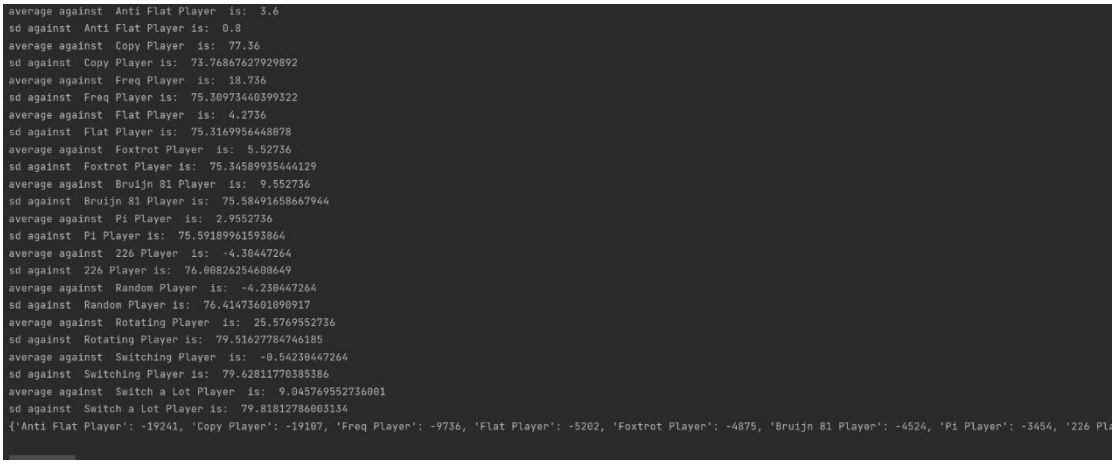
To avoid problems such as circularity we made sure not to change the most fit player unless it's better than the globally best player

Our player is stochastic since we use a lot of randomly generated values in our genetic algorithm and it learns from his opponent's moves and we don't use meta strategies but also we don't follow any constant pattern or values, partially exploitive because of the elitism because it exploits the top players and partially explores duo to our mutation function.

Q9-10

Final tournament

First of all we set a match between our player and every other bot and we got the results below, as we can see we managed to win most of the games and clearly our player is better than all of the other bots



And to further prove it we arranged a double round robin tournament in which each bot and our player played 2 games against every other player and by the end the player with the highest rating win the tournament

Our rating system was the sum of all rounds won by a player and if a player loses he loses a point

And as we can see our player had the highest rating by far and with the switch a lot in second place