AI – Lab 1

Part 1

Zinat abo hgool 206721714

Elie hadad 207931536

1)the average of the fitness

```
def calc_avg_fitness(self, population: list[Struct]):# this function calculates the average of the fitness
    totalfitness = 0
    avgfitness = 0
    for i in range(self.args.GA_POPSIZE):
        totalfitness += population[i].getFitness()
    avgfitness = totalfitness / self.args.GA_POPSIZE
    return avgfitness
```

The Standard Deviation

```
def calc_SD(self, population: list[Struct], avg):# this function calculates the standard deviation
    total = 0
    for i in range(self.args.GA_POPSIZE):
        total += abs(population[i].getFitness() - avg)**2
    sd = total / self.args.GA_POPSIZE
    result = math.sqrt(sd)
    return result
```

You can see the standard deviation and the average fitness for every generation:

```
The average of the fitness is: 23.8369140625
The standard deviation is:  24.914909188114446
Clock ticks time:  0.02698493003845215
Best:  Hello Wormd! ( 1 )
The average of the fitness is: 21.724609375
The standard deviation is:  24.060700822423318
Clock ticks time:  0.03153085708618164
Best:  Hello World! ( 0 )
The average of the fitness is: 19.94140625
The standard deviation is:  24.43531169020483
```

2) clock ticks and elapsed:

To show the time of clock ticks and elapsed we added the following lines to the run function in the main file

```
    start_qt = time.time()  # the start time of clock ticks
    start_elapsed = timer()  # the start time of elapsed
    for i in range(args1.GA_MAXITER):
        iteration_time = time.time()
        ga.calc_fitness(population)
        #bull_hits(population, args1)
        ga.sort_by_fitness(population)
```

```
        ga.mate(population, buffer)
        clock_ticks = time.time() - iteration_time
        print("Clock ticks time: ", clock_ticks)
        population, buffer = buffer, population
    elapsed_time = timer() - start_elapsed
    print("Elapsed time: ", elapsed_time)
```

This is the output you can show the values of clock ticks and elapsed time:

```
The average of the fitness is: 15.28564453125
The standard deviation is:  22.42482992233201
Clock ticks time:  0.024934053421020508
Best:  Helln World! ( 1 )
The average of the fitness is: 14.44482421875
The standard deviation is:  22.233488487429575
Clock ticks time:  0.021986722946166992
Best:  Hello World! ( 0 )
The average of the fitness is: 13.3515625
The standard deviation is:  21.91849331320686

Best string: Hello World!
found in 29 iterations out of 16384
Elapsed time:  0.6869953999999998
```

3)fitness histogram:

To implement the fitness histogram, we first normalized the fitness values of the population to be between the range (0,100).

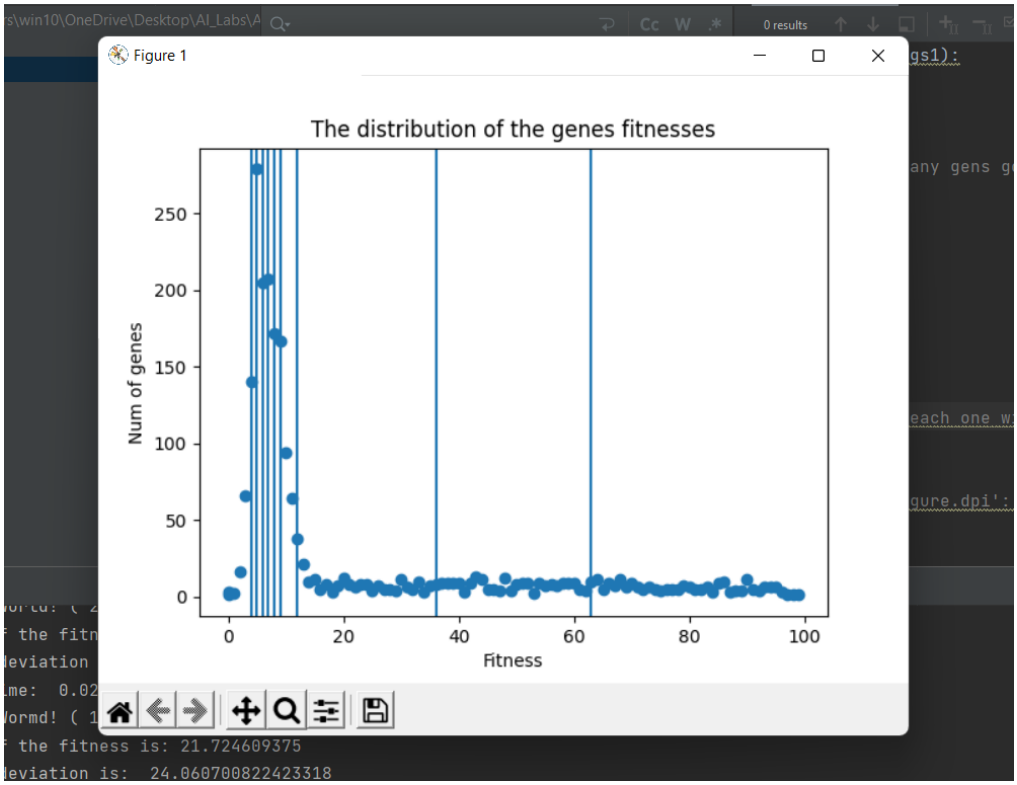
Then we stored how many genes had the same value of fitness, and in the final histogram we divided the graph by vertical lines each one of them is a quantile that represents 10% of the population size.

```python
def histogram(norm_fitness, population: list[Struct], args1):
    x = [0] * 101
    for i in range(100):
        x[i] = i
    y = [0] * 101 # array that stores in each cell how many gens got this fitness value
    for i in range(args1.GA_POPSIZE):
        index = norm_fitness[i]
        y[index] += 1

    #for i in range(100):
        #print("Y is:", y[i])

    plt.scatter(x, y)
    for i in range(1,10):# split the graph to quantiles each one with the size of 10% of popsize
       plt.axvline(x=(norm_fitness[204*i]))

    #plt.rcParams.update({'figure.figsize': (10, 8), 'figure.dpi': 100})
    plt.title('The distribution of the genes fitnesses')
    plt.xlabel('Fitness')
    plt.ylabel('Num of genes')
    plt.show()
```



4)single, two, uniform crossover:

According to what we learned in the lecture we implemented the different types of crossover as the following:

```
self.elitism(population, buffer)
for i in range(self.esize, self.args.GA_POPSIZE):
    i1 = randint(0, (self.args.GA_POPSIZE/2) - 1)
    i2 = randint(0, (self.args.GA_POPSIZE/2) - 1)
    spos = randint(0, self.tsize - 1)
    if(self.args.CROSS_TYPE == "Single"):
        buffer[i] = Struct(population[i1].str[0: spos] + population[i2].str[spos:],
    if(self.args.CROSS_TYPE =="Two"):
        spos1 = randint(0, self.tsize - 2)
        spos2 = randint(spos1 + 1, self.tsize - 1)
        buffer[i] = Struct(population[i1].str[0: spos1] + population[i2].str[spos1:
    if(self.args.CROSS_TYPE == "Uniform"):
        gene = ""
        for j in range(self.tsize):
            x =randint(0, sys.maxsize) % 2
            if x == 0:
                gene = gene + population[i1].str[j]
            else:
                gene = gene + population[i2].str[j]
        buffer[i] = Struct(gene, 0)
    if random() < self.args.GA_MUTATION:
        self.mutate(buffer[i])
```

5)Bull's eye:

We implemented heuristic function called bull_hits that based on the idea of bull's eye, it gives penalty on the fitness of the gene if the letter was not on the right place or was missing:

```
plt.show()
def bull_hits(population: list[Struct], args1): # this function implement bull heuristic that gives
    # penalty of the fitness if the letter is not in the right place or if it doesn't exist
    for i in range(args1.GA_POPSIZE):
        new_fitness = 0
        tsize = len(args1.GA_TARGET)
        for j in range(tsize):
            if population[i].str[j] == args1.GA_TARGET[j]:# if the letter in the right place don't give penalty
                new_fitness += 0
            else:
                for k in range(j+1, tsize):
                    if population[i].str[j] == args1.GA_TARGET[k]:# if the letter somewhere in the string give it some penalty
                        new_fitness += 25
                        break
                new_fitness += 50 #if the letter dosen't exist in the target string give it heavy penalty
        population[i].fitness = new_fitness
```

6)Bull's eye VS. distance of letters heuristic

We concluded that bull's eye heuristic performance is preferable over the performance of the original heuristic(distance of the letters). If we imply the different crossover type in both heuristic, we can see clearly that bull's eye brings less generation number, with less numbers of iterations to find the solution compared to original heuristic.

7)exploration VS. exploitation

Exploration: our code implemented exploration in the section of crossover, because it brings new child to the population, and by this operation the code explore new string for the new child, also we consider the part of mutate function as exploration, because in this function we change the string.

Exploitation: in the elitism function we have exploitation because in this function we concentrate on specific percent of the best genes in the population, this means that we want to stay with this specific percent and not to explore for others.

8)PSO

This is our implementation of PSO algorithm:

This first class for the particle and its attributes and functions

```
class Particle():
    def __init__(self, position, fitness, velocity):
        self.pos_size = len(position)
        self.velocity = velocity
        self.position = position# the particle's string
        self.fitness = fitness
        self.pbest_pos = position
        self.pbest_fitness = fitness

    def make_round(self, position):#function to round number to be complete
        for i in range(len(position)):
            position[i] = round(position[i])
        return position

    def move(self, gbest_pos, c1, c2, W):
            r1 = random()
            r2 = random()
            num1 = self.pso_str(self.pbest_pos, self.position, c1 * r1)
            num2 = self.pso_str(gbest_pos, self.position, c2 * r2)
            str = self.merge(num1, num2)
            self.velocity = [(self.velocity[i] * W) for i in range(len(self.velocity))]

            self.velocity = self.merge(self.velocity, str)
            self.position = self.merge(self.position, self.velocity)


            self.velocity = self.make_round(self.velocity)
            self.position = self.make_round(self.position)


    def merge(self, str1, str2):#function to merge two strings
        new_string = []
        for i in range(len(str1)):
            new_string.append((str1[i]) + (str2[i]))
        return new_string

    def pso_str(self, str1, str2, para):#function to create a new string that match pso algorithm
        new_string = []
        for i in range(len(str1)):
            new_string.append((para * ((str1[i]) - (str2[i]))))
        return new_string
```

The second class for the whole swarm that consist of the particles and other different attributes and functions

```
class Swarm():
    def __init__(self, args):
        self.args = args
        self.particles = []
        self.gbest_pos = None
        self.tsize = len(self.args.GA_TARGET)
        self.gbest_fitness = float('inf')
        self.W = 0.5
        self.c1 = 0.8
        self.c2 = 0.9

    def calc_fitness(self, particle):
        str_target = [ord(char) for char in self.args.GA_TARGET]
        fitness = 0
        for i in range(self.tsize):
            fitness += (abs(particle[i] - str_target[i]))
        return fitness

    def sort_by_fitness(self, particles):
        particles.sort(key=lambda i: i.fitness)
        self.gbest_pos = particles[0].position
        self.gbest_fitness = particles[0].fitness
```

```
def initParticles(self):#initilaize the particles array that is indeed the population or the swarm
    for i in range(self.args.GA_POPSIZE):
        velocity = []
        position = []
        for j in range(self.tsize):
            velocity.append(randint(32, 122))
            position.append(randint(32, 122))
        if self.gbest_pos is None:#initilize global best position
            self.gbest_pos = position[0:]
        elif self.calc_fitness(position) < self.calc_fitness(self.gbest_pos):
            self.gbest_pos = position
        particle = Particle(position, self.calc_fitness(position), velocity)
        self.particles.append(particle)
def to_str(self, gbest_pos):#function to convert to type string
    str = ""
    for i in range(len(gbest_pos)):
        str += chr(gbest_pos[i])
    return str

def set_pbest(self, fitness, particle):#function to get particle best
        if(particle.pbest_fitness > fitness):
            particle.pbest_fitness = fitness
            particle.pbest_pos = particle.position
```

```
def set_gbest(self, fitness, particle):#function to get global best
        if(self.gbest_fitness > fitness):
            self.gbest_fitness = fitness
            self.gbest_pos = particle.position

def update_para(self, index, size):#function to update pso parameters
    self.c1 = -3 * (index / size) + 3.5
    self.c2 = 3 * (index / size) + 0.5
    self.W = 0.4 * ((index - size)/ size ** 2) + 0.4
```

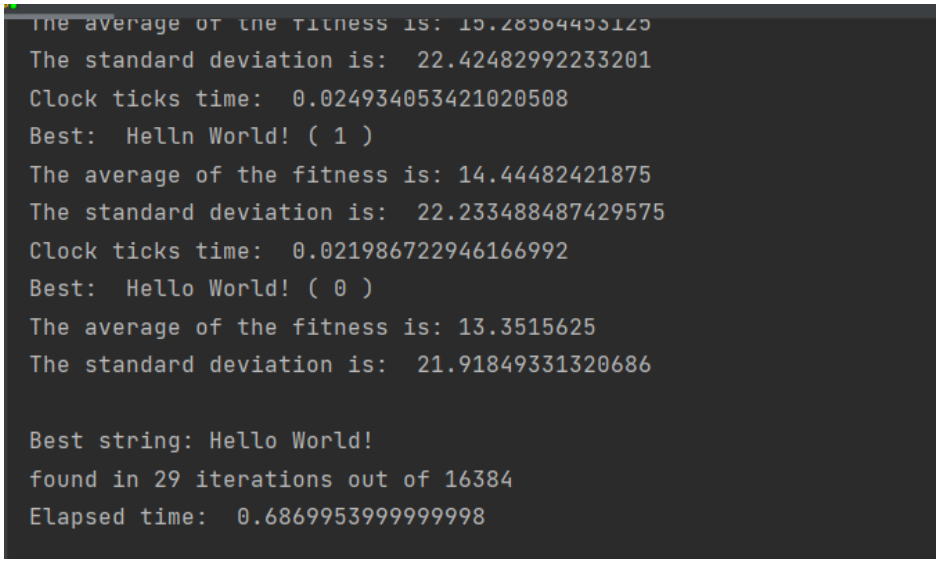
When we run the PSO algorithm we get the following output:

```
Clock ticks time:  0.039895057678222656
Best string: Hello Wosld! ( 1 )
Clock ticks time:  0.04392051696777344
Best string: Hello Wosld! ( 1 )
Clock ticks time:  0.051819562911987305
Best string: Hello Wosld! ( 1 )
Clock ticks time:  0.04790782928466797
Best string: Hello Wosld! ( 1 )
Clock ticks time:  0.045879364013671875
Best string: Hello Wosld! ( 1 )
Clock ticks time:  0.04085516929626465
Best string:  Hello World! ( 0 )
Elapsed time:  3.8438411000000006
```

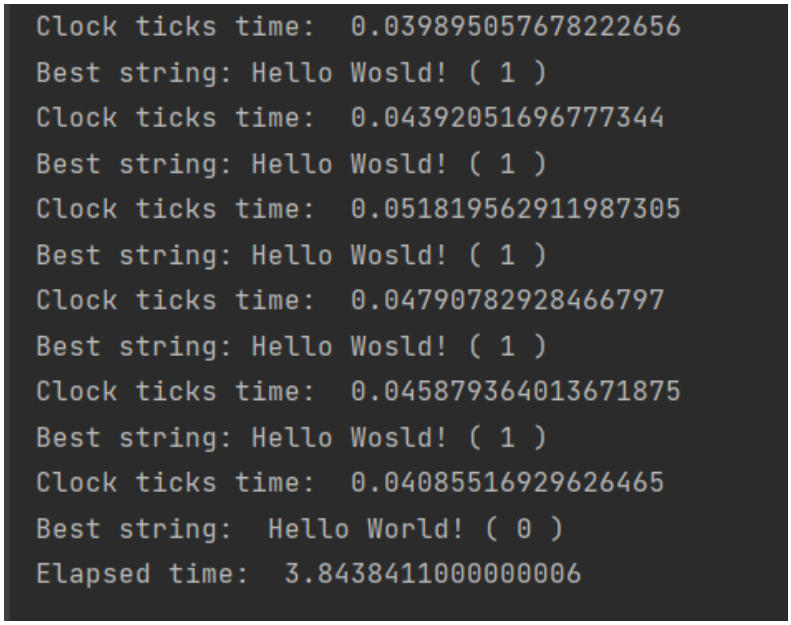
9)PSO VS. GA:

We can see clearly when we run both algorithms the difference between elapsed time, so the performance of GA is preferable over PSO

GA time:



PSO time:



Lab 1 – part 2:

1)

RWS + sigma scaling:

```
    def RWS(self, population, buffer, f=-1):
        # Return num_winners RWS Selections
        selection = self.roulette_spin(population)
        return selection

    def sigmascaling(self, population: list[Struct], i: int):
        avg = self.calc_avg_fitness(population)
        sd = self.calc_SD(population, avg)
        return 1 + (population[i].fitness - avg) / 2 * sd
```

Ruolette spin:

```
def roulette_spin(self, population: list[Struct], numberofwinners=1):
    # Given a list of fitnesses, returns the result of the roulette spin where
    # probability of choosing individual i = f(i)/sum(fitnesses)
    fitnesses: list[int] = []
    counter = 0
    for pop in population:
        fitnesses.append(pop.fitness)
        counter += 1
    fitness_sum = sum(fitnesses)
    probabilities = [fitness / fitness_sum for fitness in fitnesses]
    return np.random.choice(len(fitnesses), numberofwinners, probabilities)
```

Tournament selection:

```
def tournamentSelection(self, population):
    best = None
    index: int
    for i in range(self.args.tournamentK):
        person = population[randint(0, len(population) - 1)]
        if best is None:
            best = person
            index = i
        elif best.fitness > person.fitness:
            best = person
            index = i
    return index
```

SUS:

```
def SUS(self, population, esize):
    parent = self.args.GA_POPSIZE - esize
    fitness = [-1]
    max_fit = max(gene.fitness for gene in population)
    for i in range(len(population)):
        fitness.append(max_fit - population[i].fintess)
    fitness = numpy.array(fitness)
    total1 = fitness.total()
    total2 = total1[-1]
    forward = int(total2 / parent)
    begin = random.randrange(forward)
    selected_gene = np.arange(begin, total2, forward)
    selected_pop = np.searchsorted(total1, selected_gene)
    return [population[gene] for gene in selected_pop]
```

2)Aging:

To support the aging feature in the genetic algorithm we implemented the following:

***we selected the ideal aging range to be between 2 and 30**

```
def is_age(gene):#function to check if the gene in the appropriate range of age
    if 2 <= gene.age <= 30:
        return 1
    else:
        return 0
```

And we changed the elitism function in order to support aging:

```
def elitism(self, population: list[Struct], buffer: list[Struct]):
    flag = 0
    temp = []
    while len(temp) < self.esize:
        if is_age(population[flag]) & flag < self.args.GA_POPSIZE:
            temp.append(population[flag])
            flag += 1
        else:
            flag += 1
    #temp = population[:self.esize].copy()
    buffer[:self.esize] = temp
    for i in range(self.esize):
        buffer[i].age += 1
```

3) N-queens:

We added a file to implement N-queens problem you can check the full code to see it, here is a snip:

```
class NQueens:
    def __init__(self,args):
        self.args=args
        self.nqueens=args.nqueens


    def canQueenAttack(self,qR, qC, oR, oC):

        # If queen and the opponent are
        # in the same row
        if qR == oR:
            return True

        # If queen and the opponent are
        # in the same column
        if qC == oC:
            return True

        # If queen can attack diagonally
        if abs(qR - oR) == abs(qC - oC):
            return True

        # Opponent is safe
        return False

    def calc_fitness(self, population: list[Struct]):

        for i in range(self.args.GA_POPSIZE):
            fitness = self.nqueens
```

This is the following output when we run N-queens:

```
Best:  41357226 ( 5 )
Clock ticks time:  0.10653042793273926
Best:  41357226 ( 5 )
Clock ticks time:  0.11228036880493164
Best:  24135365 ( 5 )
Clock ticks time:  0.09817242622375488
Best:  17266434 ( 5 )
Clock ticks time:  0.09997010231018066
Best:  17266434 ( 5 )
Clock ticks time:  0.12005782127380371
Best:  16277443 ( 5 )
Clock ticks time:  0.0940084457397461
Best:  62514737 ( 4 )
Clock ticks time:  0.09914207458496094
Best:  53174222 ( 4 )
Clock ticks time:  0.10275673866271973
Best:  62751051 ( 5 )
Clock ticks time:  0.09552478790283203
```

```
Best:  51642733 ( 2 )
Clock ticks time:  0.09740328788757324
Best:  53174200 ( 3 )
Clock ticks time:  0.10729336738586426
Best:  53174202 ( 3 )
Clock ticks time:  0.10609078407287598
Best:  53174200 ( 3 )
Clock ticks time:  0.1059877872467041
Best:  53174202 ( 3 )
Clock ticks time:  0.11838865280151367
Best:  53174202 ( 3 )
Clock ticks time:  0.10669350624084473
Best:  53174602 ( 0 )

Best string: 53174602
found in 35 iterations out of 16384
Elapsed time:  4.139390800000001
```

4) applying PMX & XC & simple inverse mutate & swap mutate

PMX:

This function takes the permutations of two genes and implement the following:

```
def pmx(perm1, perm2):
    str_size = len(perm1)
    pmx = randint(0, str_size-1)
    index1 = perm1[pmx]
    index2 = perm2[pmx]
    for i in range(str_size):
        if perm1[i] == index2:
            perm1[i], perm1[pmx] = index1, perm1[i]
    for i in range(str_size):
        if perm2[i] == index1:
            perm2[i], perm2[pmx] = index2, perm2[i]

    return [perm1, perm2]
```

CX:

we based our algorithm for implementing the cycle crossover on the following article:

/https://www.hindawi.com/journals/cin/2017/7430125

also here the function take two permutations of the parents and returns the possible children as the following:

```
def cx(perm1, perm2):
    size = len(perm1)
    child1 = Struct("", 0)
    child2 = Struct("", 0)
    for i in range(size):#initilize the permutation of each  child with -1
        child1.permut[i] = -1
        child2.permut[i] = -1
    first = randint(1, 2)#choose randomly(or from the first parent or the second)  the first chromosome of the first child
    if first == 1:
       child1.permut[0] = perm1[0]
       child2.permut[0] = perm2[0]
       index = 0
       while perm2[index] != child1.permut[0]:#while we didn't complete a cycle
           for i in range(size):
               if perm1[i] == perm2[index]:
                   child1.permut[i] = perm1[i]
                   child2.permut[i] = perm2[i]
                   index = i
                   break
       for i in range(size):
           if child1.permut[i] == -1:
               child1.permut[i] = perm2[i]
               child2.permut[i] = perm1[i]
           index = 0
           while perm1[index] != child1.permut[0]:
               for i in range(size):
                   if perm2[i] == perm1[index]:
                       child1.permut[i] = perm2[i]
                       child2.permut[i] = perm1[i]
                       index = i
                       break
           for i in range(size):
               if child1.permut[i] == -1:
                   child1.permut[i] = perm1[i]
                   child2.permut[i] = perm2[i]

       return child1, child2
```

simple inverse mutate:

```
def simple_inverse_mutate(member):#inverse the string between specific range
    size = len(member.str)
    posi = randint(0, size-2)
    posj = randint(posi+1, size-1)
    if posi > posj:
        posi, posj = posj, posi
    while posi < posj:
        member.str[posi], member.str[posj] = member.str[posj], member.str[posi]
        posi += 1
        posj -= 1
```

Swap mutation:

```
def swap_mutate(member):#swaps two random indeces in the string
    size = len(member.str)
    posi = randint(0, size - 2)
    posj = randint(posi + 1, size - 1)
    str1 = ""
    str1 = member.str[0:posi] + member.str[posj] + member.str[posi + 1:posj]+ member.str[posi] + member.str[posj+1:]
    member.str = str1
```

5) N-queens VS. Bull's eye:

	N-queens	Bull's eye
Population size	The more the population size is bigger we get more run time and less generated genes, and vice versa	The more the population size is bigger we get more run time and less generated genes, and vice versa
Mutate probability	When we increase the mutation probability, we get less generated genes and less run time, vice versa	When we increase the mutation rate, generally we get we get less generated genes and less run time, vice versa(in some cases we get slightly more)
Selection strategy	The fastest strategy was tournament, comparing to sus and rws	The fastest strategy was tournament, comparing to sus and rws
Survival strategy	We choose the perfect age to be between 2 to 30, although survival strategy based on fitness was preferable	We choose the perfect age to be between 2 to 30, although survival strategy based on fitness was preferable
Crossover type	We get better performance when we applied CX rather than PMX	We get better performance when we applied uniform corssover

***the optimal collection of parameters that we want to use in applying the next sections is:**

POP_SIZE = 1800

MUTATE_RATE = 0.5

ELITRATE = 0.1

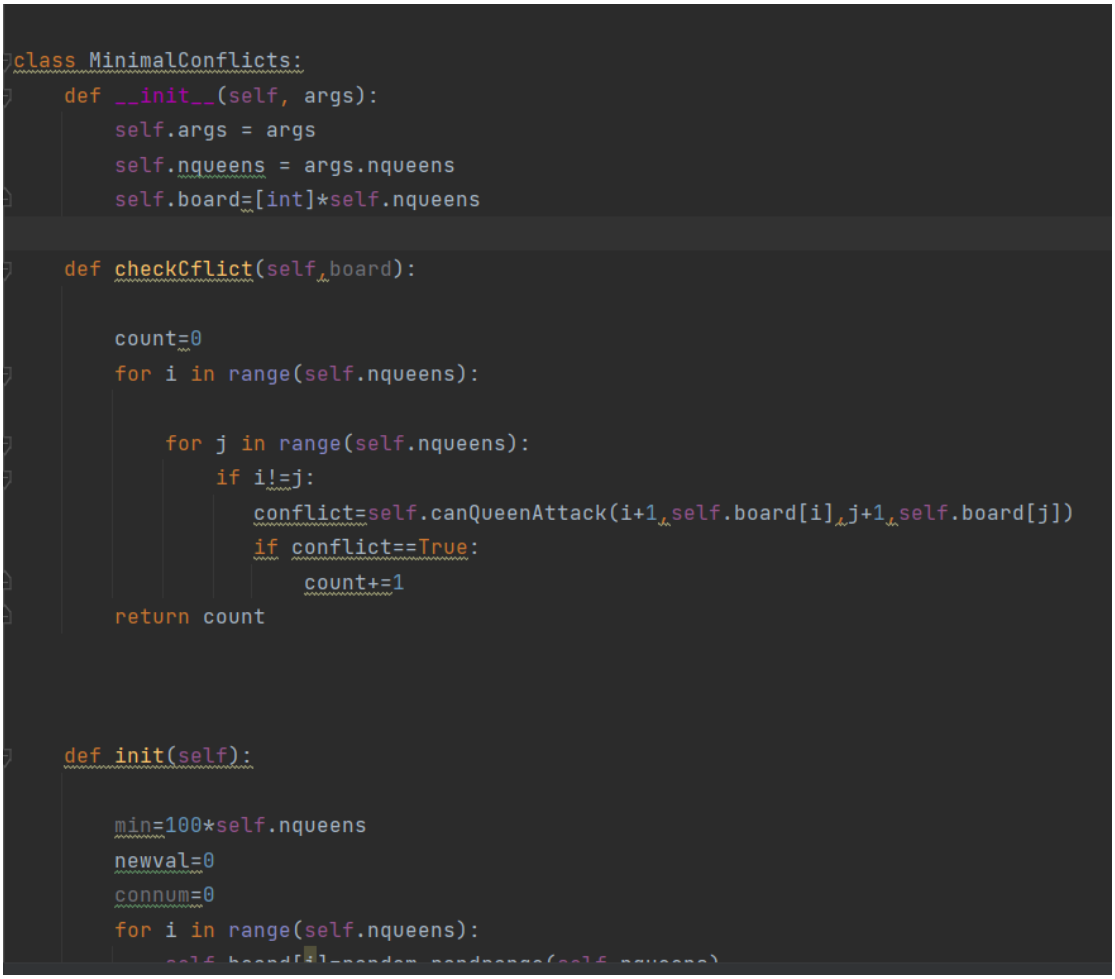
CROSSOVER_TYPE = uniform(for bull's eye)/ CX (for N-queens)

SELECTION_TYPE = tournament

NUM_QUEENS = 8

6)minimal conflicts:

We added new file that support the implementation of minimal conflicts, here a snip of it:



```
class MinimalConflicts:
    def __init__(self, args):
        self.args = args
        self.nqueens = args.nqueens
        self.board=[int]*self.nqueens

    def checkCflict(self,board):

        count=0
        for i in range(self.nqueens):

            for j in range(self.nqueens):
                if i!=j:
                    conflict=self.canQueenAttack(i+1,self.board[i],j+1,self.board[j])
                    if conflict==True:
                        count+=1
        return count



    def init(self):

        min=100*self.nqueens
        newval=0
        connum=0
        for i in range(self.nqueens):
```

7)Bin packing

To implement the bin packing algorithm we added files, here some of the code that support it:

```python
class Bins:
    def __init__(self, arr, fitness):
        self.arr = arr
        self.fitness = fitness

    def getString(self):
        return self.arr

    def getFitness(self):
        return self.fitness

    def setString(self, arr):
        self.arr = arr

    def setFitness(self, fitness):
        self.fitness = fitness
```

```python
from Bins import Bins
import random
import sys
class BinPacking:
    def __init__(self,args):
        self.args=args
        self.tsize=len(args.weightarr)
        self.esize=int(self.args.GA_POPSIZE * self.args.GA_ELITRATE)
        self.numBins=args.numBins
        self.weightarr=args.weightarr
        self.binsize=args.binsize

    def initpop(self,population: list[Bins], buffer: list):
        for i in range(self.args.GA_POPSIZE):
            temp=[int]*self.tsize
            fitness1 = 0
            for j in range(self.tsize):
                temp[j] = random.randrange(self.numBins)

            population[i]=Bins(temp,fitness1)
```