CSE422 Lab Assignment 2

Md. Shamiul Islam

ID: 17301108

Theory Section: 8

Lab Section: 4

Task 1

import numpy as np

#initial population

def genRanPop(size\_of\_population):

for i in range(size\_of\_population):

ranPop = np.random.randint(0, nQ,nQ)

# print(ranPop)

pList.append(ranPop)

return pList

#fitness function

def aPair(SingleParent,SizePop):

attackingPair=0

for refQueen in range(SizePop):

for comQueen in range(refQueen+1,SizePop):

if SingleParent[refQueen]==SingleParent[comQueen]:

attackingPair+=1

return attackingPair

def fitFunction(ParentList,populationSize):

non\_attackingPair=populationSize\*(populationSize-1)//2

for parent in ParentList:

net\_non\_attackingPair=non\_attackingPair-aPair(list(parent),populationSize)

pFit.append(net\_non\_attackingPair)

return pFit

#selection function

def select\_Parent(parentContainer, parentFit):

sumScore=sum(parentFit)

i=0

for score in parentFit:

parentFit[i]=round(score/sumScore,2)

i+=1

chromosome\_1=list(parentFit).index(max(list(parentFit)))

parentFit[list(parentFit).index(max(list(parentFit)))]=0

chromosome\_2=list(parentFit).index(max(list(parentFit)))

return parentContainer[chromosome\_1],parentContainer[chromosome\_2]

#crossover function

def crossoverChildren(c1, c2):

c1=list(c1)

c2=list(c2)

off1=np.array(c1[0:len(c1)//2]+c2[len(c2)//2:])

off2=np.array(c2[0:len(c2)//2]+c1[len(c1)//2:])

return off1,off2

#mutation function

def mutateOffspring(child\_to\_mulate):

print("\nBefore Mutation: ")

print(child\_to\_mulate)

queenFlip=np.random.randint(0,nQ,1)

child\_to\_mulate[queenFlip[0]]=np.random.randint(0,nQ,1)[0]

print("\nAfter Mutation: ")

return child\_to\_mulate

#genetic algorithm

def GA(populationDatabase, nP,factor):

fitness=fitFunction(population,nP)

child1,child2=select\_Parent(population, pFit)

print("\nBefore Performing Crossover: \n",child1,"\n",child2)

offspring1,offspring2=crossoverChildren(child1,child2)

print("\nAfter Performing Crossover: \n",offspring1,"\n",offspring2)

#new population

populationDatabase=[]

populationDatabase.append(offspring1)

populationDatabase.append(offspring2)

newFit=fitFunction(populationDatabase,nP)

total\_fitValue=newFit[-1] + newFit[-2]

newFit[-1]=round(newFit[-1]/ total\_fitValue,2)

newFit[-2]=round(newFit[-2]/total\_fitValue,2)

newParent=[]

if newFit[-1]>=newFit[-2]:

if newFit[-1]>factor:

populationDatabase[1]=mutateOffspring(populationDatabase[1])

print(populationDatabase[1])

else:

if newFit[-2]>factor:

populationDatabase[0]=mutateOffspring(populationDatabase[0])

print(populationDatabase[0])

return populationDatabase

#number of queens

nQ = int(input('Enter the no. of Queens: '))

#population size

pSize = int(input("Enter the \'Population Size\' :"))

#population carrier list

pList = []

#initial population

population=genRanPop(pSize)

print("\nInitial Population: ")

parent\_serial=1

for participator in population:

print("Serial no-",parent\_serial," : ",participator)

parent\_serial+=1

itr=int(input("\nEnter the Iteration: \n"))

i=1

while itr>0:

#fitness score

pFit=[]

mutation\_threshold = 0.3

print("\nIteration: ",i,end=" : ")

population=GA(population, nQ, mutation\_threshold)

i+=1

itr-=1

Task 2

#installing deap

!pip install deap

import numpy as np

import matplotlib.pyplot as plt

from deap import algorithms, base, benchmarks, cma, creator, tools

# create a toolbox

def create\_toolbox(strategy):

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register("evaluate", benchmarks.rastrigin)

# Seed random number generator

np.random.seed(7)

toolbox.register("generate", strategy.generate, creator.Individual)

toolbox.register("update", strategy.update)

return toolbox

if \_\_name\_\_ == "\_\_main\_\_":

# problem\_size

num\_individuals = 10

num\_generations = 125

# create a strategy using CMA-ES algorithm

strategy = cma.Strategy(centroid=[5.0]\*num\_individuals, sigma=5.0, lambda\_=20\*num\_individuals)

# create toolbox based on the above strategy

toolbox = create\_toolbox(strategy)

# create hall of fame object

hall\_of\_fame = tools.HallOfFame(1)

# register the relevant stats

stats = tools.Statistics(lambda x: x.fitness.values)

stats.register("avg", np.mean)

stats.register("std", np.std)

stats.register("min", np.min)

stats.register("max", np.max)

logbook = tools.Logbook()

logbook.header = "gen", "evals", "std", "min", "avg", "max"

# objects that will compile the data

sigma = np.ndarray((num\_generations, 1))

axis\_ratio = np.ndarray((num\_generations, 1))

diagD = np.ndarray((num\_generations, num\_individuals))

fbest = np.ndarray((num\_generations,1))

best = np.ndarray((num\_generations, num\_individuals))

std = np.ndarray((num\_generations, num\_individuals))

for gen in range(num\_generations):

# generate a new population

population = toolbox.generate()

# evaluate the individuals

fitnesses = toolbox.map(toolbox.evaluate, population)

for ind, fit in zip(population, fitnesses):

ind.fitness.values = fit

# strategy with the evaluated

toolbox.update(population)

# recent evaluated population

hall\_of\_fame.update(population)

record = stats.compile(population)

logbook.record(evals=len(population), gen=gen, \*\*record)

print(logbook.stream)

# data along the evolution for plot

sigma[gen] = strategy.sigma

axis\_ratio[gen] = max(strategy.diagD)\*\*2/min(strategy.diagD)\*\*2

diagD[gen, :num\_individuals] = strategy.diagD\*\*2

fbest[gen] = hall\_of\_fame[0].fitness.values

best[gen, :num\_individuals] = hall\_of\_fame[0]

std[gen, :num\_individuals] = np.std(population, axis=0)

# x-axis

x = list(range(0, strategy.lambda\_ \* num\_generations, strategy.lambda\_))

avg, max\_, min\_ = logbook.select("avg", "max", "min")

plt.figure()

plt.semilogy(x, avg, "--b")

plt.semilogy(x, max\_, "--b")

plt.semilogy(x, min\_, "-b")

plt.semilogy(x, fbest, "-c")

plt.semilogy(x, sigma, "-g")

plt.semilogy(x, axis\_ratio, "-r")

plt.grid(True)

plt.title("Blue: f-values, Green: sigma, Red: axis ratio")

plt.figure()

plt.plot(x, best)

plt.grid(True)

plt.title("object variables")

plt.figure()

plt.semilogy(x, diagD)

plt.grid(True)

plt.title("scaling ")

plt.figure()

plt.semilogy(x, std)

plt.grid(True)

plt.title("SD in all coordinates")

plt.show()