**DAA CIA 2 ASSIGNMENT**

RUSHIKESH MURAHARISETTY

Reg.no. 21011102067

Class: IOTB

GITHUB LINK: https://github.com/zenyuzien/DAA\_CIA2

1)

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define XC 500

#define YC 500

#define NO\_CITIES 50

#define POP 1500

#define ITERATIONS 5000

#define MUTATION\_P 0.25

#define PRINT\_BY 150

typedef struct

{

unsigned x, y;

}city;

typedef struct

{

int solution[NO\_CITIES];

double fitness;

}individual;

void swap(int \* a, int \* b){

int t = \*a;

\*a = \*b;

\*b = t;

}

int highest\_fitness(individual \* population){

int index = 0;

double initial = population[0].fitness;

for (int j = 1; j < POP; j++){

if (population[j].fitness > initial){

initial = population[j].fitness;

index = j;

}

}

return index;

}

void setup\_cities(city \* cities\_map){

for (int i = 0; i < NO\_CITIES; i++){

cities\_map[i].x = rand() % XC;

cities\_map[i].y = rand() % YC;

}

}

void generate\_initial\_population(individual \* population){

int x;

for (int i = 0; i < POP; i++){

for (int j = 0; j < NO\_CITIES; j++){

x = rand() % (j+1);

if (x != j){

population[i].solution[j] = population[i].solution[x];

}

population[i].solution[x] = j;

}

}

}

double get\_distance(city \* cities\_map, int id\_1, int id\_2){

int x1 = cities\_map[id\_1].x;

int y1 = cities\_map[id\_1].y;

int x2 = cities\_map[id\_2].x;

int y2 = cities\_map[id\_2].y;

double dist = sqrt((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2));

return dist;

}

double \* evaluate\_fitness(city \* cities\_map, individual \* population){

int id\_1, id\_2;

double total, fitness\_sum = 0;

double \* cdf = malloc(sizeof(double) \* POP);

for (int i = 0; i < POP; i++){

total = 0;

for (int j = 0; j < NO\_CITIES-1; j++){

id\_1 = population[i].solution[j];

id\_2 = population[i].solution[j+1];

total += get\_distance(cities\_map, id\_1, id\_2);

}

population[i].fitness = 1/total;

fitness\_sum += population[i].fitness;

}

cdf[0] = population[0].fitness/fitness\_sum;

for (int i = 1; i < POP; i++){

cdf[i] = cdf[i-1] + population[i].fitness/fitness\_sum;

}

return cdf;

}

int \* selection(double \* cdf){

double extraction;

int pivot, first, last;

int \* candidates = malloc(sizeof(int) \* POP);

for (int i = 0; i < POP; i++){

first = 0;

last = POP;

extraction = (double)rand()/RAND\_MAX;

while (first != last){

pivot = (first + last)/2;

if (cdf[pivot] == extraction){

break;

}

if (cdf[pivot] > extraction){

last = pivot;

} else {

first = pivot + 1;

}

}

candidates[i] = first;

}

free(cdf);

return candidates;

}

void cycle\_crossover(individual \* population, int id\_1, int id\_2, int \* child\_1, int \* child\_2){

int parent\_1 = id\_1, parent\_2 = id\_2;

int j = 0;

int lookup[NO\_CITIES];

int flags[NO\_CITIES] = {0};

int visited = 0;

int first = population[id\_1].solution[0], last;

for (int i = 0; i < NO\_CITIES; i++){

lookup[population[id\_1].solution[i]] = i;

}

while (visited < NO\_CITIES){

child\_1[j] = population[parent\_1].solution[j];

child\_2[j] = population[parent\_2].solution[j];

last = population[id\_2].solution[j];

flags[j] = 1;

visited++;

j = lookup[last];

if (first == last){

swap(&parent\_1, &parent\_2);

while (flags[j]) j++;

first = population[id\_1].solution[j];

}

}

}

individual \* crossover(individual \* population, int \* candidates){

individual \* new\_gen = malloc(sizeof(individual) \* POP);

for (int i = 0; i < POP; i+=2){

cycle\_crossover(population, candidates[i], candidates[i+1], new\_gen[i].solution, new\_gen[i+1].solution);

}

free(candidates);

return new\_gen;

}

void mutation(individual \* population){

int x, y, p = RAND\_MAX \* MUTATION\_P;

for (int i = 0; i < POP; i++){

if (rand() < p){

do {

x = rand() % NO\_CITIES;

y = rand() % NO\_CITIES;

} while (x == y);

swap(&population[i].solution[x], &population[i].solution[y]);

}

}

}

int main()

{

srand(time(NULL));

city cities\_map[NO\_CITIES];

individual \* population, \* new\_gen;

population = malloc(sizeof(individual) \* POP);

setup\_cities(cities\_map);

generate\_initial\_population(population);

int best;

int \* candidates;

double length;

double \* cdf;

for (int i = 0; i < ITERATIONS; i++)

{

cdf = evaluate\_fitness(cities\_map, population);

if (i%PRINT\_BY == 0)

{

best = highest\_fitness(population);

length = 1/population[best].fitness;

printf("the new generation's (gen.no %d) best (fit) length is %f\n", i, length);

}

candidates = selection(cdf);

new\_gen = crossover(population, candidates);

free(population);

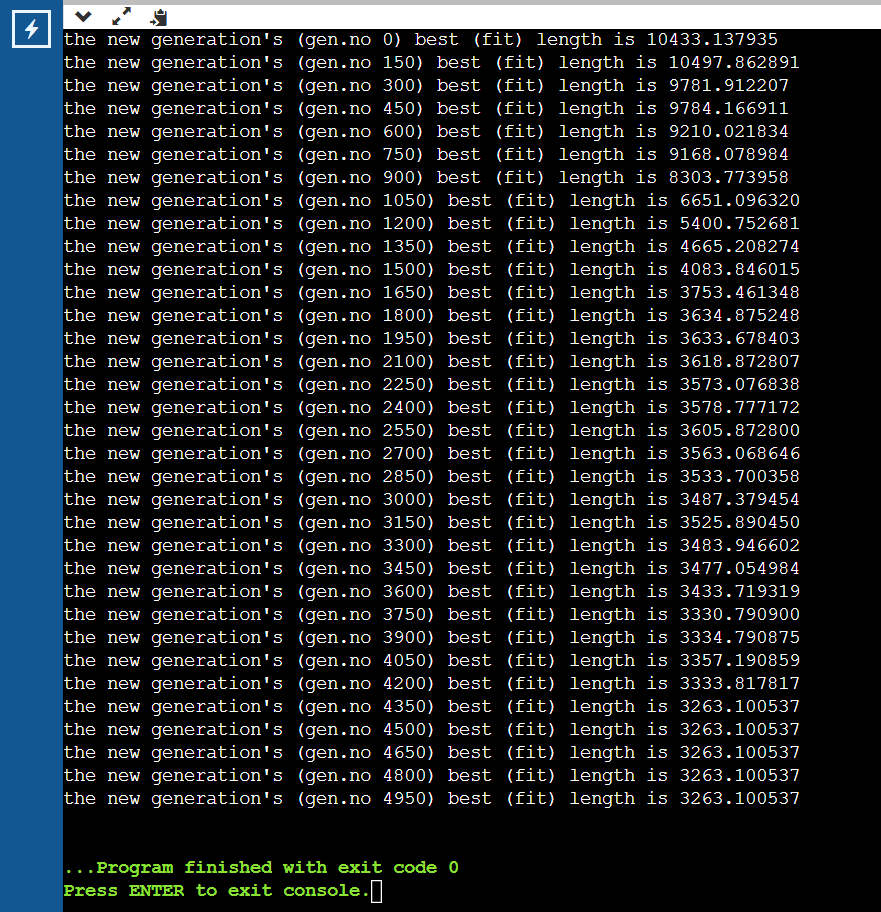
population = new\_gen;

mutation(population);

}

return 0;

}



2)

import numpy as np

import matplotlib.pyplot as plt

no\_population = 75

num\_generations = 125

rate\_of\_mutation = 0.02

selection\_size = 15

population = np.random.uniform(low=0.0, high=2\*np.pi, size=(no\_population,))

def fitness\_f(x):

return np.sum(np.sin(x))

for generation in range(num\_generations):

fitness = np.array([fitness\_f(x) for x in population])

selected\_population = population[np.argsort(fitness)[-selection\_size:]]

children = np.empty((no\_population,))

for i in range(no\_population):

parent1, parent2 = np.random.choice(selected\_population, size=2, replace=False)

children[i] = np.random.uniform(low=0.0, high=2\*np.pi) if np.random.rand() < rate\_of\_mutation \

else (parent1 + parent2) / 2

population = children

best\_ind = population[np.argmax(np.array([fitness\_f(x) for x in population]))]

best\_fit = fitness[np.argmax(np.array([fitness\_f(x) for x in population]))]

x = np.linspace(0, 2\*np.pi, 1000) ; y = np.sin(x)

plt.plot(x, y, label='True function')

y = np.sin(best\_ind)

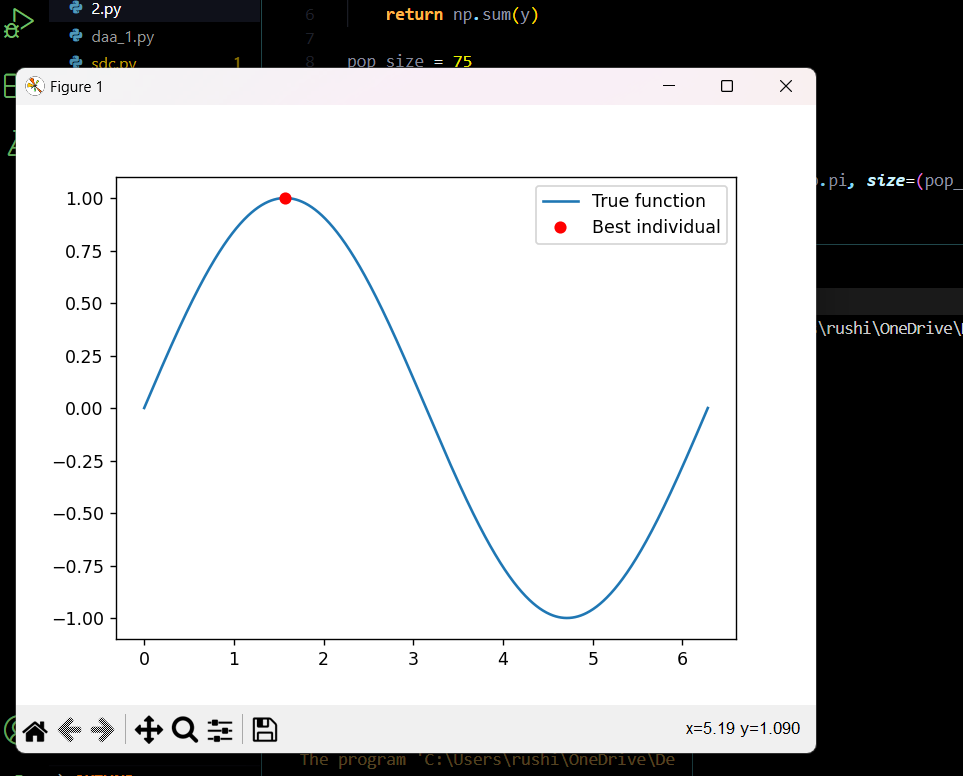
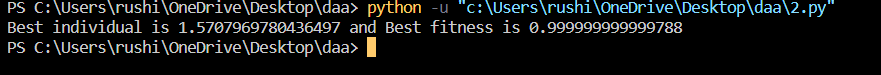
plt.plot(best\_ind, y, 'ro', label='Best individual')

plt.legend()

plt.show()

# printing result

print(f'Best individual is {best\_ind} and Best fitness is {best\_fit}')

3)

import numpy as np

import matplotlib.pyplot as plt

def fx(x):

y = np.sin(x)

return np.sum👍

population = 75

iterations = 125

rate\_of\_mutation = 0.02

selection = 15

cultural\_rate = 0.15

pool\_size = 7

population = np.random.uniform(low=0.0, high=2\*np.pi, size=(population,))

cultural\_pool = population[np.argsort(np.array([fx(x) for x in population]))[-pool\_size:]]

for generation in range(iterations):

fitness = np.array([fx(x) for x in population])

selected\_population = population[np.argsort(fitness)[-selection:]]

for i in range(population):

if np.random.rand() < cultural\_rate:

cultural\_individual = np.random.choice(cultural\_pool)

population[i] = cultural\_individual

children = np.empty((population,))

for i in range(population):

parent1, parent2 = np.random.choice(selected\_population, size=2, replace=False)

children[i] = np.random.uniform(low=0.0, high=2\*np.pi) if np.random.rand() < rate\_of\_mutation \

else (parent1 + parent2) / 2

population = children

fitness = np.array([fx(x) for x in population])

cultural\_pool = population[np.argsort(fitness)[-pool\_size:]]

fitness = np.array([fx(x) for x in population])

best\_individual = population[np.argmax(fitness)]

best\_fitness = fitness[np.argmax(fitness)]

x = np.linspace(0, 2\*np.pi, 1000) ; y = np.sin(x)

plt.plot(x, y, label='True function')

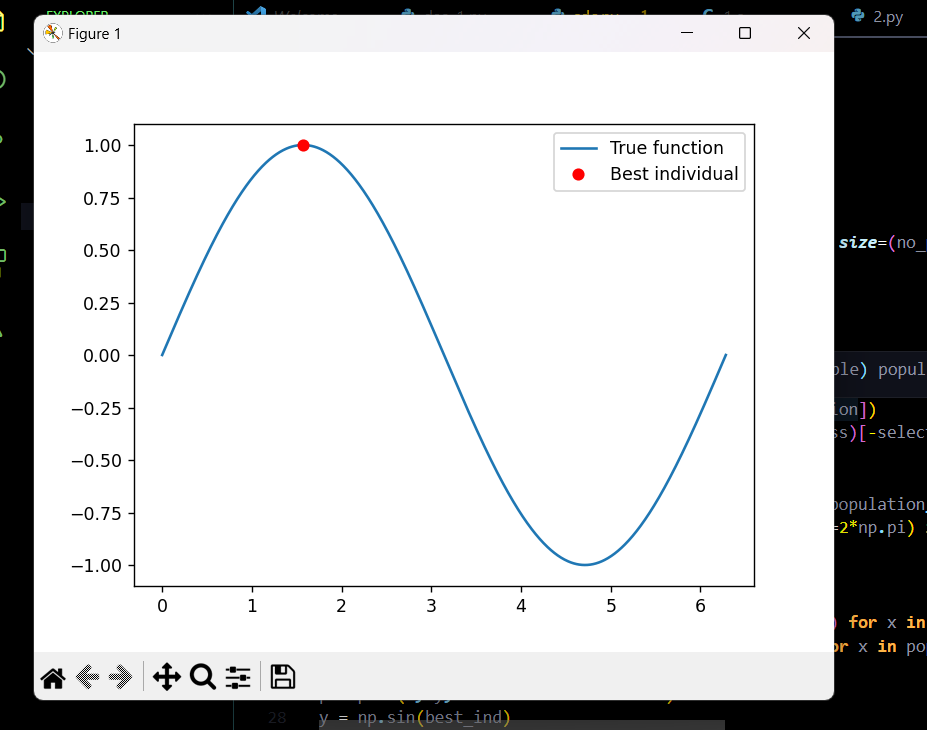
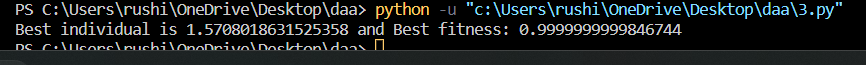
y = np.sin(best\_individual)

plt.plot(best\_individual, y, 'ro', label='Best individual')

plt.legend()

plt.show()

print(f'Best individual is {best\_individual} and Best fitness: {best\_fitness}')

4)

import random

import math

def sine(y):

return math.sin👍

#PSO algorithm

def psox(func, gen=125, nparts=75, range=(-15, 15), c1=1, c2=1, w=0.5):

particles = []

for i in range(nparts):

particle = {'pos': random.uniform(range[0], range[1]),'velocity': 0,'best\_pos': None,'bscore': None}

particle['best\_pos'] = particle['pos']

particle['bscore'] = func(particle['pos'])

particles.append(particle)

global\_best\_pos = particles[0]['best\_pos']

gbscore = particles[0]['bscore']

for i in range(gen):

for particle in particles:

particle['velocity'] = w \* particle['velocity'] \

+ c1 \* random.random() \* (particle['best\_pos'] - particle['pos']) \

+ c2 \* random.random() \* (global\_best\_pos - particle['pos'])

particle['pos'] += particle['velocity']

if particle['pos'] < range[0]:

particle['pos'] = range[0]

elif particle['pos'] > range[1]:

particle['pos'] = range[1]

score = func(particle['pos'])

if score > particle['bscore']:

particle['bscore'] = score

particle['best\_pos'] = particle['pos']

if score > gbscore:

gbscore = score

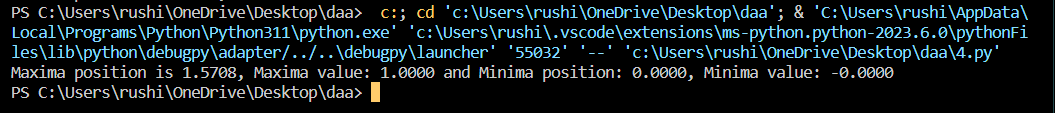
global\_best\_pos = particle['pos']

return global\_best\_pos, gbscore

maxima\_pos, maxima\_val = psox(sine, range=(0, math.pi), nparts=100)

minima\_pos, minima\_val = psox(lambda x: -sine(x), range=(0, math.pi), nparts=100)

print(f"Maxima pos is {maxima\_pos:.4f}, Maxima value: {maxima\_val:.4f} and Minima pos: {minima\_pos:.4f}, Minima value: {minima\_val:.4f}")



5)

import random

import numpy as np

dist\_mat = np.array([[0, 30, 3, 3, 1],

[23, 0, 16, 4, 4],

[14, 6, 0, 18, 2],

[2, 5, 3, 0, 13],

[4, 9, 1, 13, 0]])

ants = 5 ; its = 15

evap\_rate = 0.5

alpha = 1 ; beta = 2

Q = 100

mat = np.ones(dist\_mat.shape) / len(dist\_mat)

def next\_city(present\_city, mat, dist\_mat, alpha, beta):

attractive = mat[present\_city]\*\*alpha \* (1 / dist\_mat[present\_city])\*\*beta

attractive[present\_city] = 1

probabilities = attractive / attractive.sum()

next\_city = np.random.choice(range(len(dist\_mat)), p=probabilities)

return next\_city

for iteration in range(its):

ant\_routes = []

for ant in range(ants):

start\_city = random.randint(0, len(dist\_mat) - 1)

ant\_route = [start\_city]

while len(ant\_route) < len(dist\_mat):

present\_city = ant\_route[-1]

next\_city = next\_city(present\_city, mat, dist\_mat, alpha, beta)

ant\_route.append(next\_city)

ant\_routes.append(ant\_route)

delta\_mat = np.zeros(dist\_mat.shape)

for ant\_route in ant\_routes:

for i in range(len(ant\_route) - 1):

start\_city = ant\_route[i]

end\_city = ant\_route[i + 1]

delta\_mat[start\_city][end\_city] += Q / dist\_mat[start\_city][end\_city]

delta\_mat[end\_city][start\_city] += Q / dist\_mat[end\_city][start\_city]

mat = (1 - evap\_rate) \* mat + delta\_mat

best\_ant\_route = min(ant\_routes, key=lambda x: sum([dist\_mat[x[i]][x[i+1]] for i in range(len(x) - 1)]))

print(f"Current generation {iteration}, ant route genearted : {best\_ant\_route}")

