**Design and analysis of algorithms**

**CIA 2**

G.S. Ruparani

21011102032

IoT-A

Language used - Python

1. Genetic Algorithm for Travelling Salesman Problem (TSP)

import random

import numpy as np

]n\_cities = 10

cities = np.random.rand(n\_cities, 2)

pop\_size = 100

n\_generations = 100

mutation\_rate = 0.1

def create\_chromosome(n\_cities):

cities = list(range(n\_cities))

random.shuffle(cities)

return cities

def evaluate\_fitness(chromosome, cities):

dist = 0

for i in range(len(chromosome)):

j = (i + 1) % len(chromosome)

city\_i, city\_j = chromosome[i], chromosome[j]

dist += np.linalg.norm(cities[city\_i] - cities[city\_j])

return dist

population = [create\_chromosome(n\_cities) for \_ in range(pop\_size)]

for generation in range(n\_generations):

fitness = [evaluate\_fitness(chromosome, cities) for chromosome in population]

idx = np.argsort(fitness)

selected\_population = [population[i] for i in idx[:pop\_size//2]]

new\_population = []

while len(new\_population) < pop\_size:

parent1, parent2 = random.sample(selected\_population, 2)

child = parent1.copy()

start, end = sorted(random.sample(range(n\_cities), 2))

for i in range(start, end + 1):

if parent2[i] not in child[start:end+1]:

idx = child.index(parent2[i])

child[i], child[idx] = child[idx], child[i]

new\_population.append(child)

for i in range(len(new\_population)):

if random.random() < mutation\_rate:

idx1, idx2 = random.sample

1. Cultural Algorithm for TSP

import random

import numpy as np

# Define the problem

n\_cities = 10

cities = np.random.rand(n\_cities, 2)

# Define the CA parameters

n\_cultures = 5

pop\_size = 20

n\_generations = 100

mutation\_rate = 0.1

# Define the chromosomes

def create\_chromosome(n\_cities):

cities = list(range(n\_cities))

random.shuffle(cities)

return cities

# Define the fitness function

def evaluate\_fitness(chromosome, cities):

dist = 0

for i in range(len(chromosome)):

j = (i + 1) % len(chromosome)

city\_i, city\_j = chromosome[i], chromosome[j]

dist += np.linalg.norm(cities[city\_i] - cities[city\_j])

return dist

# Initialization

populations = [[] for \_ in range(n\_cultures)]

for i in range(n\_cultures):

populations[i] = [create\_chromosome(n\_cities) for \_ in range(pop\_size)]

for generation in range(n\_generations):

# Cultural exchange

for i in range(n\_cultures):

source = i

target = random.randint(0, n\_cultures-1)

if target == source:

continue

best\_chromosome = min(populations[source], key=lambda x: evaluate\_fitness(x, cities))

populations[target].append(best\_chromosome)

# Selection

new\_populations = [[] for \_ in range

1. Genetic Algorithm for Sine Function

import random

import math

def sine\_function(x, amplitude, frequency, phase):

return amplitude \* math.sin(frequency \* x + phase)

def find\_maximum\_sine\_value(lower\_limit, upper\_limit, population\_size, num\_generations, mutation\_rate):

# Define the chromosomes

def create\_chromosome():

amplitude = random.uniform(0, 1)

frequency = random.uniform(0, 1)

phase = random.uniform(0, math.pi)

return [amplitude, frequency, phase]

def fitness(chromosome):

max\_value = -math.inf

for x in range(lower\_limit, upper\_limit + 1):

y = sine\_function(x, chromosome[0], chromosome[1], chromosome[2])

if y > max\_value:

max\_value = y

return max\_value

population = [create\_chromosome() for i in range(population\_size)]

for generation in range(num\_generations):

# Evaluate the fitness of each chromosome

fitness\_values = [fitness(chromosome) for chromosome in population]

def selection(fitness\_values):

sum\_fitness = sum(fitness\_values)

probabilities = [fitness\_value / sum\_fitness for fitness\_value in fitness\_values]

return random.choices(population, probabilities, k=2)

def crossover(parents):

offspring = [0, 0, 0]

for i in range(3):

offspring[i] = (parents[0][i] + parents[1][i]) / 2

return offspring

def mutation(chromosome):

for i in range(3):

if random.random() < mutation\_rate:

chromosome[i] = random.uniform(0, 1) if i < 2 else random.uniform(0, math.pi)

return chromosome

offspring = []

for i in range(population\_size):

parents = selection(fitness\_values)

child = crossover(parents)

child = mutation(child)

offspring.append(child)

offspring\_fitness\_values = [fitness(chromosome) for chromosome in offspring]

combined\_population = population + offspring

combined\_fitness\_values = fitness\_values + offspring\_fitness\_values

population = [combined\_population[i] for i in random.choices(range(len(combined\_population)), combined\_fitness\_values, k=population\_size)]

fitness\_values = [fitness(chromosome) for chromosome in population]

max\_fitness\_index = fitness\_values.index(max(fitness\_values))

return population[max\_fitness\_index], max(fitness\_values)

lower\_limit = 0

upper\_limit = 10

population\_size = 20

num\_generations = 50

mutation\_rate = 0.1

best\_chromosome, max\_fitness\_value = find\_maximum\_sine\_value(lower\_limit, upper\_limit, population\_size, num\_generations, mutation\_rate)

print("Best chromosome:", best\_chromosome)

print("Maximum fitness value:", max\_fitness\_value)

1. Particle Swarm Optimization for the sine function

import random

import math

def sine\_function(x, amplitude, frequency, phase):

return amplitude \* math.sin(frequency \* x + phase)

def find\_extreme\_sine\_values(lower\_limit, upper\_limit, swarm\_size, num\_iterations, w, c1, c2):

def create\_particle():

amplitude = random.uniform(0, 1)

frequency = random.uniform(0, 1)

phase = random.uniform(0, math.pi)

velocity = [random.uniform(-1, 1) for i in range(3)]

return [amplitude, frequency, phase, velocity, amplitude, frequency, phase]

def fitness(chromosome):

max\_value = -math.inf

min\_value = math.inf

for x in range(lower\_limit, upper\_limit + 1):

y = sine\_function(x, chromosome[0], chromosome[1], chromosome[2])

if y > max\_value:

max\_value = y

chromosome[4] = max\_value

chromosome[5] = chromosome[0]

chromosome[6] = chromosome[1]

if y < min\_value:

min\_value = y

chromosome[7] = min\_value

chromosome[8] = chromosome[0]

chromosome[9] = chromosome[1]

return max\_value - min\_value

swarm = [create\_particle() for i in range(swarm\_size)]

global\_best\_particle = [0, 0, 0, [0, 0, 0], -math.inf, 0, 0, math.inf, 0, 0]

for iteration in range(num\_iterations):

# Evaluate the fitness of each particle

for particle in swarm:

particle\_fitness = fitness(particle)

# Update the personal best for each particle

if particle\_fitness > particle[4] - particle[7]:

particle[4] = particle\_fitness + particle[7]

particle[5] = particle[0]

particle[6] = particle[1]

if particle\_fitness > global\_best\_particle[4] - global\_best\_particle[7]:

global\_best\_particle = list(particle)

for particle in swarm:

for i in range(3):

particle[3][i] = w \* particle[3][i] + c1 \* random.uniform(0, 1) \* (particle[5+i] - particle[i]) + c2 \* random.uniform(0, 1) \* (global\_best\_particle[5+i] - particle[i])

particle[i] += particle[3][i]

# Ensure the values are within bounds

if i < 2:

particle[i] = max(0, min(1, particle[i]))

else:

particle[i] = max(0, min(math.pi, particle[i]))

return (global\_best\_particle[5:8], global\_best\_particle[8:])

1. Ant Colony Optimization for finding the shortest path in a graph.

import random

import numpy as np

class Graph:

def \_\_init\_\_(self, num\_nodes):

self.num\_nodes = num\_nodes

self.edges = np.zeros((num\_nodes, num\_nodes))

def add\_edge(self, i, j, distance):

self.edges[i][j] = distance

self.edges[j][i] = distance

class Ant:

def \_\_init\_\_(self, start\_node):

self.start\_node = start\_node

self.path = [start\_node]

self.visited\_nodes = set([start\_node])

self.distance\_travelled = 0

def move\_to\_next\_node(self, probabilities):

next\_node = random.choices(range(len(probabilities)), probabilities)[0]

self.path.append(next\_node)

self.visited\_nodes.add(next\_node)

self.distance\_travelled += graph.edges[self.path[-2]][next\_node]

def can\_move\_to\_node(self, node):

return node not in self.visited\_nodes

def find\_shortest\_path\_in\_graph(graph, num\_ants, evaporation\_rate, alpha, beta, num\_iterations):

pheromone\_trails = np.ones((graph.num\_nodes, graph.num\_nodes))

np.fill\_diagonal(pheromone\_trails, 0)

for iteration in range(num\_iterations):

# Generate ants and let them move through the graph

ants = [Ant(i) for i in range(graph.num\_nodes)]

for ant in ants:

for i in range(graph.num\_nodes - 1):

current\_node = ant.path[-1]

unvisited\_nodes = [node for node in range(graph.num\_nodes) if ant.can\_move\_to\_node(node)]

if not unvisited\_nodes:

break

pheromone\_values = np.power(pheromone\_trails[current\_node][unvisited\_nodes], alpha)

distance\_values = np.power(1 / graph.edges[current\_node][unvisited\_nodes], beta)

probabilities = pheromone\_values \* distance\_values / np.sum(pheromone\_values \* distance\_values)

ant.move\_to\_next\_node(probabilities)

ant.distance\_travelled += graph.edges[ant.path[-1]][ant.start\_node]

pheromone\_trails \*= (1 - evaporation\_rate)

for ant in ants:

for i in range(graph.num\_nodes - 1):

pheromone\_trails[ant.path[i]][ant.path[i+1]] += 1 / ant.distance\_travelled

pheromone\_trails[ant.path[-1]][ant.start\_node] += 1 / ant.distance\_travelled

shortest\_path = None

shortest\_distance = np.inf

for ant in ants:

if ant.distance\_travelled < shortest\_distance:

shortest\_path = ant.path

shortest\_distance = ant.distance\_travelled

return shortest\_path, shortest\_distance