**DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING**

**COLLEGE OF E&ME, NUST, RAWALPINDI**

AI & Decision Support Systems

Lab Report #6

**Student Name: Nawab Aarij Imam**

**Degree/ Syndicate: 43 CE - A**

**Task:**

**Code:**

import numpy as np

def generate\_population(num\_chromosomes):

return np.random.randint(8, size=(num\_chromosomes, 8))

def calculate\_fitness(chromosome):

non\_attacking\_pairs = 28

for i in range(8):

for j in range(i+1, 8):

if chromosome[i] == chromosome[j] or abs(chromosome[i] - chromosome[j]) == abs(i - j):

non\_attacking\_pairs -= 1

return non\_attacking\_pairs

def calculate\_fitness\_percentages(chromosomes):

fitnesses = np.apply\_along\_axis(calculate\_fitness, axis=1, arr=chromosomes)

return fitnesses / np.sum(fitnesses)

def select\_parents(chromosomes):

fitness\_percentages = calculate\_fitness\_percentages(chromosomes)

cumulative\_probabilities = np.cumsum(fitness\_percentages)

selected\_pairs = []

for \_ in range(len(chromosomes)):

parents = []

while len(parents) < 2:

rand = np.random.random()

parent\_index = np.argwhere(cumulative\_probabilities > rand)[0][0]

if parent\_index not in parents:

parents.append(parent\_index)

selected\_pairs.append(parents)

return selected\_pairs

def crossover\_and\_mutate(parent1, parent2):

crossover\_point = np.random.randint(0, 8)

child = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

mutation\_gene = np.random.randint(0, 8)

child[mutation\_gene] = np.random.randint(0, 8)

return child

def evolve\_population(chromosomes):

new\_population = []

parent\_pairs = select\_parents(chromosomes)

for pair in parent\_pairs:

new\_chromosome = crossover\_and\_mutate(chromosomes[pair[0]], chromosomes[pair[1]])

new\_population.append(new\_chromosome)

return np.array(new\_population)

def solve\_eight\_queens(population\_size, max\_generations):

population = generate\_population(population\_size)

for generation in range(max\_generations):

fitnesses = np.apply\_along\_axis(calculate\_fitness, axis=1, arr=population)

if generation % 1000 == 0:

print(f'Generation {generation}: Max fitness = {np.max(fitnesses)}')

if 28 in fitnesses:

solution\_index = np.where(fitnesses == 28)[0][0]

return population[solution\_index], generation

population = evolve\_population(population)

return None, max\_generations

def main():

population\_size = 20

max\_generations = 10000

solution, generations = solve\_eight\_queens(population\_size, max\_generations)

if solution is not None:

print(f'\nSolution found in generation {generations}:')

print(solution)

print('\nBoard representation:')

for row in range(8):

board\_row = ['Q' if solution[row] == col else '.' for col in range(8)]

print(' '.join(board\_row))

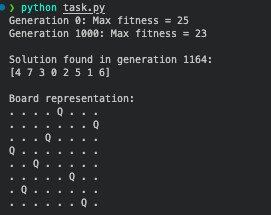
else:

print('\nNo solution found within the maximum number of generations.')

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

main()

**Output:**

****