COE 9

Lab Evaluation: Assignment 1

1. Solve the following **8-puzzle problem using genetic algorithm**.

Initial state:

5	1	8
2	6	3
7	0	4

Goal State

5	2	8
1	6	4
7	0	3

The number of misplaced tiles in the current state as compared to the goal state is to be considered as the fitness function. Apply **single-point crossover** followed by **two-bit flip mutation** for offspring generation. Also show how many iterations it took to reach goal state?

CODE:

```
import numpy as np
import random as rd
from random import randint

# start state as 1d array
initial_arr = [5,1,8,2,6,3,7,0,4]
# goal state as 1d array
final_arr = [5,2,8,1,6,4,7,0,3]

# population size
solutions_per_pop = 4
# population size as a 2d array
pop_size = (solutions_per_pop, len(initial_arr))
print('Population size = {}'.format(pop_size))

# initial population (here I repeated same initial array 4 times as
mutation will eventually evolve the population)
initial_population = np.array([[5,1,8,2,6,3,7,0,4], [5,1,8,2,6,3,7,0,4],
[5,1,8,2,6,3,7,0,4], [5,1,8,2,6,3,7,0,4]])
print('Initial population: \n{}'.format(initial_population))


# heuristic function which helps in evaluating the fitness of each
chromosome as the number of misplaced tiles
# when compared with the goal state
def heuristic(initial_arr, final_arr):
    # count represents the total number of mismatch
    count = 0
    for p in range(9):
        if initial_arr[p]!=0:
            # whenever there is a mismatch in the current state and goal
state it is added to the count
            if initial_arr[p]!=final_arr[p]:
                count = count+1
    return count


# cal_fitness function which calculates the fitness of population at each
iteration
```

```
def cal_fitness(final_arr, population):
    # fitness is a 1d array which has the fitness value as no of misplaced
tiles for each chromosome of the population
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        # for each chromosome we find the fitness value using heuristic
function
        # lower the fitness value, better the solution
        fitness[i] = heuristic(population[i], final_arr)
    return fitness.astype(int)


# selection function which is used to select parents for crossover and
mutation based on their fitness value
def selection(fitness, num_parents, population):
    fitness = list(fitness)
    # parents will contain the chromosomes which are selected as fittest,
and further evolve by crossover and mutation
    parents = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
        # here lower the fitness value better the solution
        max_fitness_idx = np.where(fitness == np.min(fitness))
        parents[i,:] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = +99999
    return parents


# crossover function performs the 1-point crossover of the parents to
generate offsprings for the next generation
def crossover(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    # here we will perform 1-point crossover from the middle of the
chromosome
    crossover_point = int(parents.shape[1] / 2)
    # crossover_rate will define how frequent we want crossover to happen
    crossover_rate = 0.8
    x = rd.random()
    # here we compare the crossover_rate with a random value,
    # if it is greater than the random value, then only we perform
crossover,
    # otherwise we return the parents as it is
    if x > crossover_rate:
        return parents
    # first half of the first offspring as it is from the first parent
    offsprings[0, 0:crossover_point] = parents[0, 0:crossover_point]
    j=0
    k=0
    for i in parents[1]:
        if i not in parents[0, 0:crossover_point]:
            # remaining half of the first offspring from the second parent
in orderly fashion
            offsprings[0, crossover_point+j] = i
            j += 1
        else:
            # first half of the second offspring from the remaining
elements of the second parent in orderly fashion
            offsprings[1,k] = i
            k += 1
    # second half of the second offspring as it is from the second half of
the first parent
    offsprings[1, crossover_point:] = parents[0, crossover_point:]
```

```
    return offsprings


# mutation function which performs 2-bit flip mutation of the offsprings
generated after crossover
def mutation(offsprings):
    # mutation rate will define how frequent we want mutation to happen
    mutation_rate = 0.4
    mutants = np.empty((offsprings.shape))
    for i in range(mutants.shape[0]):
        random_value = rd.random()
        mutants[i, :] = offsprings[i, :]
        # here we compare mutation rate with a random value,
        # if mutation rate is higher than the random value, then only
perform mutation,
        # otherwise keep offsprings as it is
        if random_value > mutation_rate:
            continue
        # here we select two random bits for 2-bit flip mutation
        int_random_value1 = randint(0, offsprings.shape[1] - 1)
        int_random_value2 = randint(0, offsprings.shape[1] - 1)
        # here we flip the bits
        temp = mutants[i][int_random_value1]
        mutants[i][int_random_value1] = mutants[i][int_random_value2]
        mutants[i][int_random_value2] = temp
    return mutants


# optimize function handles all the operations to generate a new generation
# which takes the population closer to the goal state
def optimize(final_arr, population, pop_size):
    final_state = []
    # here we consider half the population as number of parents
    num_parents = int(pop_size[0] / 2)
    # remaining population will be formed by the offsprings from these
parents
    num_offsprings = pop_size[0] - num_parents
    # iter represents the number of iteration taken to reach the goal state
    iter = 0

    # we keep iterating till goal state is achieved
    while True:
        # calculating fitness of the population
        fitness = cal_fitness(final_arr, population)
        # checking if goal state is achieved
        if np.min(fitness) == 0:
            break
        # selecting fittest chromosomes as parents
        parents = selection(fitness, num_parents, population)
        # generating offsprings from the above parents
        offsprings = crossover(parents, num_offsprings)
        # performing mutation of the above offsprings
        mutants = mutation(offsprings)
        # we keep the parents as it is in the population
        population[0:parents.shape[0], :] = parents
        # we keep the newly generated offsprings in the population,
discarding the previous unfit chromosomes
        population[parents.shape[0]:, :] = mutants
        iter = iter + 1
```

```
    print("\nIterations are:", iter)
    # printing the last generation and its fitness
    print('Last generation: \n{}\n'.format(population))
    fitness_last_gen = cal_fitness(final_arr, population)
    print('Fitness of the last generation:
\n{}\n'.format(fitness_last_gen))
    # the chromosome which will be the fittest is actually our goal state
    max_fitness = np.where(fitness_last_gen == np.min(fitness_last_gen))
    final_state.append(population[max_fitness[0][0], :])
    # we return the goal state and the number of iterations
    return final_state, iter


# calling the optimize function
final_state, iter = optimize(final_arr, initial_population, pop_size)
# printing the result
print('The optimized current state is: \n{}\nachieved in {}
iterations.'.format(final_state, iter))
```

OUTPUT:

```
Run:  LabEval_assign1_q1
C:\Users\kulpr\PycharmProjects\OpenCVpython\venv\Scripts\python.exe C:/Users/k
Population size = (4, 9)
Initial population:
[[5 1 8 2 6 3 7 0 4]
 [5 1 8 2 6 3 7 0 4]
 [5 1 8 2 6 3 7 0 4]
 [5 1 8 2 6 3 7 0 4]]

Iterations are: 31
Last generation:
[[5 1 8 2 6 4 7 0 3]
 [5 1 8 2 6 4 7 0 3]
 [5 2 8 1 6 4 7 0 3]
 [5 1 8 4 6 2 7 0 3]]

Fitness of the last generation:
[2 2 0 3]

The optimized current state is:
[array([5, 2, 8, 1, 6, 4, 7, 0, 3])]
achieved in 31 iterations.

Process finished with exit code 0
```

2. Missionaries and Cannibals is a problem in which 3 missionaries and 3 cannibals want to cross from the left bank of a river to the right bank of the river. There is a boat on the left bank, but it only carries at most two people at a time (and can never cross with zero people). If cannibals ever outnumber missionaries on either bank, the cannibals will eat the missionaries. Solve this problem using DFS.

CODE:

```
import copy

# state representation
# [missionaries on left, cannibals on left, side of river]
initial_state   = [3,3,0]   # 0 means boat is on left side
goal_state      = [0,0,1]   # 1 means boat is on right side

# all possible moves
possible_moves = [(2,0), (0,2), (1,1), (1,0), (0,1)]
# move's first element represent number of missionaries on the boat
# move's second element represent number of cannibals on the boat

# negative sign represents boat going from left to right
# positive sign represents boat going from right to left
sign = (-1, +1)      # direction

stack = []  # stack to implement DFS
visited = []    # list to keep track of all visited states
stack.append(initial_state)  # initial state added to stack initially
count = 0   # No. of moves in DFS to attain goal state


# compare function which compares the states
# to determine whether or not goal state has been reached
def compare(arr, goal_state):
    for i in range(len(arr)):
        if arr[i] != goal_state[i]:
            return False
    return True


# isValid function determines whether a given state is valid or not
def isValid(arr):
    # all missionaries on left side of river
    if arr[0] == 3 and ((arr[1] >= 0) and arr[1] <= 3):
        return True
    # missionaries and cannibals in equal number on both sides
    elif arr[0] == arr[1] and ((arr[1] >= 0) and arr[1] <= 3):
        return True
    # all missionaries on right side of the river
    elif arr[0] == 0 and ((arr[1] >= 0) and arr[1] <= 3):
        return True
    # otherwise not a valid move
    else:
        return False


# genChildren function generates children of a given state
# by performing all valid moves
def genChildren(arr):
    children = []
    # sign determines the direction of move
    # based on the current position of the boat
    s = sign[arr[2]]
    for move in possible_moves:
        temp = copy.deepcopy(arr)
        # here we perform all possible moves
        temp[0] = temp[0] + (s * move[0])
        temp[1] = temp[1] + (s * move[1])
```

```
        if temp[2] == 0:
            temp[2] = 1
        else:
            temp[2] = 0
        # here we check if the performed move is valid or not
        if isValid(temp):
            # all valid moves generates states
            # if those states have not yet been visited,
            # we add that state to the children of current state
            if temp not in visited and temp not in stack:
                children.append(temp)
    return children


# dfs() performs the Depth First Search
def dfs():
    global count
    # iterate till stack is not empty
    while len(stack)!=0:
        # by default it pops the element from the end,
        # i.e. most recently added element
        arr = stack.pop()
        # add this state to the visited[]
        visited.append(copy.deepcopy(arr))
        # check if the goal state has been reached
        if compare(arr,goal_state):
            print(f"No. of Moves: {count}")
            print("Path:")
            for i in visited:
                print(i)
            break
        else:
            # generate children of the current state
            child = genChildren(arr)
            # add the children into the stack
            for c in child:
                stack.append(copy.deepcopy(c))
        count += 1

if __name__ == '__main__':
    dfs()
```

OUTPUT: