Name: Danish Baloch

Roll-number: 21k-4829

Task=01:

```python
import random
POPULATION_SIZE =1000
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''
TARGET = "Artifical Intelligence Lab"

class Individual(object):

  def __init__(self, chromosome):
    self.chromosome = chromosome
    self.fitness = self.cal_fitness()

  @classmethod
  def mutated_genes(self):

    global GENES
    gene = random.choice(GENES)
    return gene

  @classmethod
  def create_gnome(self):

    global TARGET
    gnome_len = len(TARGET)
    return [self.mutated_genes() for _ in range(gnome_len)]
```

```
def mate(self, par2):


  child_chromosome = []
  for gp1, gp2 in zip(self.chromosome, par2.chromosome):

    prob = random.random()
    if prob < 0.45:
      child_chromosome.append(gp1)


    elif prob < 0.90:
      child_chromosome.append(gp2)

    else:
      child_chromosome.append(self.mutated_genes())


  return Individual(child_chromosome)

def cal_fitness(self):

  global TARGET
  fitness = 0
```

```
    for gs, gt in zip(self.chromosome, TARGET):
      if gs != gt: fitness+= 1
    return fitness

def main():
  global POPULATION_SIZE

  generation = 1

  found = False
  population = []

  for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

  while not found:
    population = sorted(population, key = lambda x:x.fitness)
    if population[0].fitness <= 0:
      found = True
      break
    new_generation = []
    s = int((10*POPULATION_SIZE)/100)
    new_generation.extend(population[:s])
    s = int((90*POPULATION_SIZE)/100)
```

```python
    for _ in range(s):
      parent1 = random.choice(population[:50])
      parent2 = random.choice(population[:50])
      child = parent1.mate(parent2)
      new_generation.append(child)

    population = new_generation

    print("Generation: {}\tString: {}\tFitness: {}".      format(generation,
      "".join(population[0].chromosome),
      population[0].fitness))

    generation += 1


  print("Generation: {}\tString: {}\tFitness: {}".    format(generation,
    "".join(population[0].chromosome),
    population[0].fitness))

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

Output:

```
Generation: 1   String: 9k-1[pcD ?O&lx7ln%4nBZd%jo      Fitness: 23
Generation: 2   String: ok$A[kc !]B&Gx7lR%4nHN $.b      Fitness: 21
Generation: 3   String: m?@xV&:B  Int#lso.iBie eap      Fitness: 18
Generation: 4   String: Alkif,cwQ InSy.e ]e}@e Soy      Fitness: 16
Generation: 5   String: Aknificaa InBO.9 % ncx npb      Fitness: 13
Generation: 6   String: hC[if.cal I6tell8%UnLZ Mab      Fitness: 11
Generation: 7   String: UCKifCcal I$telli%enLe ;ab      Fitness: 8
Generation: 8   String: ACtifical I6 elli%encP Lab      Fitness: 5
Generation: 9   String: ACtifical I6 elli%encP Lab      Fitness: 5
Generation: 10  String: AMtifical Intell8gsnce Lab      Fitness: 3
Generation: 11  String: A{tifical Intelligence Lab      Fitness: 1
Generation: 12  String: A{tifical Intelligence Lab      Fitness: 1
Generation: 13  String: A{tifical Intelligence Lab      Fitness: 1
Generation: 14  String: Artifical Intelligence Lab      Fitness: 0
```

Task=02:

```python
import random

cities = ["A", "B", "C", "D"]
distances = {
    ("A", "B"): 5,
    ("A", "C"): 3,
    ("A", "D"): 8,
    ("B", "C"): 6,
    ("B", "D"): 7,
    ("C", "D"): 4
}

# Define the genetic algorithm parameters
POPULATION_SIZE = 50
NUM_GENERATIONS = 100
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.2

# Define the solution representation
def create_permutation():
    return random.sample(cities, len(cities))

# Define the fitness function
```

```python
def calculate_distance(solution):
    distance = 0
    for i in range(len(solution)):
        city1 = solution[i]
        city2 = solution[(i + 1) % len(solution)]
        distance += distances[(city1, city2)]
    return distance

def calculate_fitness(solution):
    distance = calculate_distance(solution)
    return 1 / distance if distance > 0 else float('inf')

# Define the genetic operators
def tournament_selection(population, tournament_size):
    tournament = random.sample(population, tournament_size)
    return max(tournament, key=lambda x: x["fitness"])

def partially_mapped_crossover(parent1, parent2):
    point1 = random.randint(0, len(parent1) - 1)
    point2 = random.randint(point1 + 1, len(parent1))
    offspring1 = parent1[:]
    offspring2 = parent2[:]
    for i in range(point1, point2):
        index1 = offspring1.index(parent2[i])
        index2 = offspring2.index(parent1[i])
```

```python
            offspring1[i], offspring1[index1] = offspring1[index1], offspring1[i]
            offspring2[i], offspring2[index2] = offspring2[index2], offspring2[i]
    return offspring1, offspring2

def swap_mutation(solution):
    index1, index2 = random.sample(range(len(solution)), 2)
    solution[index1], solution[index2] = solution[index2], solution[index1]
    return solution

# Initialize the population
population = [{"solution": create_permutation()} for i in range(POPULATION_SIZE)]

# Main loop
for generation in range(NUM_GENERATIONS):
    # Evaluate the fitness of each solution
    for individual in population:
        individual["fitness"] = calculate_fitness(individual["solution"])

    # Select the parents for the next generation
    parents = []
    for i in range(POPULATION_SIZE):
        parent1 = tournament_selection(population, 3)
        parent2 = tournament_selection(population, 3)
        parents.append((parent1, parent2))
```

```python
    # Crossover
    offspring = []
    for parent1, parent2 in parents:
        if random.random() < CROSSOVER_RATE:
            child1, child2 = partially_mapped_crossover(parent1["solution"], parent2["solution"])
            offspring.append({"solution": child1})
            offspring.append({"solution": child2})

    # Mutation
    for individual in offspring:
        if random.random() < MUTATION_RATE:
            individual["solution"] = swap_mutation(individual["solution"])

    # Evaluate the fitness of the offspring
    for individual in offspring:
        individual["fitness"] = calculate_fitness(individual["solution"])

    # Select the survivors for the next generation
    population += offspring
    population = sorted(population, key=lambda x: x["fitness"], reverse=True)
    population += offspring
    population = population[:POPULATION_SIZE]

# Print the best solution in each generation
```

```
best_solution = population[0]["solution"]
best_fitness = population[0]["fitness"]
print(f"Generation {generation}: {best_solution} ({best_fitness})")
```

Task=03:

```
class Node:
    def __init__(self,data,level,fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):

        x,y = self.find(self.data,'_')

        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):

        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
```

```python
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None


    def copy(self,root):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j
```

```python
class Puzzle:
    def __init__(self,size):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self,start,goal):
        return self.h(start.data,goal)+start.level

    def h(self,start,goal):
        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp
```

```python
def process(self):
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()

    start = Node(start,0,0)
    start.fval = self.f(start,goal)
    self.open.append(start)
    print("\n\n")
    while True:
        cur = self.open[0]
        print("")
        print("  | ")
        print("  | ")
        print(" \\\'/ \n")
        for i in cur.data:
            for j in i:
                print(j,end=" ")
            print("")
        if(self.h(cur.data,goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
```

```python
                    self.open.append(i)
                self.closed.append(cur)
                del self.open[0]

                self.open.sort(key = lambda x:x.fval,reverse=False)


puz = Puzzle(3)
puz.process()
```

Output:

```
Enter the start state matrix

1      2          3
----      4                    6
7            5                 8
Enter the goal state matrix

1         2           3
4                5                      6
7                8               ----



  |
  |
 \'/

1      2          3
----      4                    6
7            5                 8
-------------------------------------------------------
```