Lab-1

Mangal
Date 3/10/24
Page 01

Genetic Algorithm for Optimization Problems

```
import random
POPULATION_SIZE = 100
GENES = ''' abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
MNOPQRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''
TARGET = "BEST!":

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.extended(population[:s])
        s = int(90 * POPULATION_SIZE) / 100)
        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: %2d \t String: %s \t Fitness: %2d"%(generation, "".join(population[0].chromosome)
                population[0].fitness))
            generation += 1

            print("Generation: %2d \t String: %s \t
        Fitness: %2d"%(generation, "".join(population[0].
        chromosome), population[0].fitness))

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

Output:

```
gen: 1    Str: BR6CP    fitness: 3
gen: 2    Str: BR6CP    fitness: 3
gen: 3    Str: BR6CP    fitness: 3
gen: 4    Str: G?SCE    fitness: 2
gen: 5    Str: G?SCE    fitness: 2
gen: 6    Str: B#SCE    fitness: 1
gen: 7    Str: B#SCE    fitness: 1
gen: 8    Str: B#SCE    fitness: 1
gen: 9    Str: BMSCE    fitness: 0
```

Lab-2

Mangal
Date 24/10/24
Page 4

Practice 3d Swarm optimization for function optimization:

```
import random
import math
import copy
import sys

def fitness_rastrigin(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi*xi) - (10*math.cos(2*
                    math.pi*xi)) + 10
    return fitnessVal

def fitness_sphere(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi*xi)
    return fitnessVal

class particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [0.0 for i in range(dim)]
        self.velocity = [0.0 for i in range(dim)]
        self.best_part_pos = [0.0 for i in range(dim)]
        for i in range(dim):
            self.position[i] = ((maxx - minx)*
                self.rnd.random() + minx)
```

Mangal
Date
Page 5

```
        self.velocity[i] = ((maxx - minx) * self.rnd.random() +
                            minx)
        self.fitness = fitness(self.position)
        self.best_part_pos = copy.copy(self.position)
        self.best_part_fitnessVal = self.fitness

def pso(fitness, max_iter, n, dim, minx, maxx):
    w = 0.729
    c1 = 1.49445
    c2 = 1.49445

    rnd = random.Random(0)

    swarm = [Particle(fitness, dim, minx, maxx, i)
             for i in range(n)]

    best_swarm_pos = [0.0 for i in range(dim)]
    best_swarm_fitnessVal = sys.float_info.max

    for i in range(n):
        if swarm[i].fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = swarm[i].fitness
            best_swarm_pos = copy.copy(swarm[i].
                position)

    Iter = 0
    while Iter < max_iter:
        if Iter % 10 == 0 and Iter > 1:
            print("Iter = {}; best fitness = {}".
                format(Iter, best_swarm_fitnessVal))
        for i in range(n):
            for k in range(dim):
```

```
r1 = rnd.random()
r2 = rnd.random()

swarm[i].velocity[k] = (
    w * swarm[i].velocity[k] +
    c1 * r1 * (swarm[i].best_part_pos[k]
    - swarm[i].position[k]) +
    c2 * r2 * (best_swarm_pos[k] - swarm[i]
    .position[k])
)

if swarm[i].velocity[k] < minx:
    swarm[i].velocity[k] = minx
elif swarm[i].velocity[k] > maxx:
    swarm.velocity[k] = maxx

for k in range(dim):
    swarm[i].position[k] += swarm[i].velocity[k]

swarm[i].fitness = fitness(swarm[i].position)

if swarm[i].fitness < swarm[i].best_part_fitness:
    swarm[i].best_part_fitnessVal = swarm[i].fitness
    swarm[i].best_part_pos = copy.copy(swarm[i].
    position)

if swarm[i].fitness < best_swarm_fitnessVal:
    best_swarm_fitnessVal = swarm[i].fitness
    best_swarm_pos = copy.copy(swarm[i].
    position)
i += 1
t += 1
return best_swarm_pos
```

Date
Page 7

```
print("\n Begin particle swarm optimization on Rastrigin
      function\n")
dim = 3
fitness = fitness_rastrigin
print(f" Goal is to minimize Rastrigin's function in {dim}
      variable")
print(" function has known min = 0.0 at ", end = "")
print(", ".join(["0"] * (dim - 1)) + "; 0)")

num_particles = 50
max_iter = 100

print(f" Setting num_particles = {num_particles}")
print(f" Setting max_iter = {max_iter}")
print("\n Starting PSO algorithm\n")

best_position = pso(fitness, max_iter, num_particles,
  dim, -10.0, 10.0)
print("\n PSO completed\n")
print("\n Best solution found:")
print([f" {best_position[k]:.6f}" for k in range(dim)])
fitnessVal = fitness(best_position)
print(f" Fitness of best solution = {fitnessVal:.6f}")
print("\n End particle swarm for Rastrigin function\n")

print(f" Goal is to minimize Sphere function in {dim}
  variables")
print("function has known min = 0.0 at ", end = " ")
print(", ".join(["0"] * (dim - 1)) + "; 0)")
num_particles = 50
max_iter = 100
```

```
print(f"Setting num_particles = {num_particles}")
print(f"Setting max_iter = {max_iter}")
print("\n Starting PSO algorithm\n")

best_position = pso(fitness, max_iter, num_particle, dim,
                    -10.0, 10.0)
print("\n PSO Completed\n")
print("\n Best solution found:")
print([f"{best_position[k]:.6f}" for k in range(
    (dim))])
fitness_val = fitness(best_position)
print(f"Fitness of best solution = {fitness_val:.6f}")
print("\n End practical Cuckoo for Sphere function\n")
```

o/p

```
Best Solution found [0.00004, -0.00001, 0.0000]
Best solution: 0.00001
```

Lab-3

Date 14/11/24
Page 9

Ant Colony optimization for the Travelling Salesman Problem.

```
import random
import numpy as np
import math

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        return math.sqrt((self.x - city.x) ** 2 + (self.y - city.y) ** 2)

class ACO_TSP:
    def __init__(self, cities, num_ants, num_iterations, alpha=1.0, beta=2.0, rho=0.5, q0=0.9):
        self.cities = cities
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q0 = q0
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities))
        self.heuristic = np.zeros((self.num_cities, self.num_cities))
        self.best_tour = None
```

```
self.best_tour_length = float('inf')

for i in range(self.num_cities):
    for j in range(i+1, self.num_cities):
        dist = cities[i].distance(cities[j])
        self.heuristic[i][j] = 1.0 / dist if dist != 0
        else 0
        self.heuristic[j][i] = self.heuristic[i][j]

def select_next_city(self, current_city,
visited_cities):
    probabilities = np.zeros(self.num_cities)
    total_pheromone = 0.0

    for city in range(self.num_cities):
        if city not in visited_cities:
            pheromone = self.pheromone[current_city][city]
            ** self.alpha
            heuristic = self.heuristic[current_city][city]
            self.beta
            probabilities[city] = pheromone * heuristic
            total_pheromone += probabilities[city]

    if total_pheromone == 0:
        return random.choice([city for city in range
(self.num_cities) if city not in visited_cities])
    probabilities /= total_pheromone
    if random.random() < self.q0:
        next_city = np.argmax(probabilities)
    else:
        next_city = np.random.choice(self.num_cities, p=probabili
```

```
def update_pheromone(self, ants):
    self.pheromone *= (1 - self.rho)
    for ant in ants:
        pheromone_deposit = 1.0 / ant.tour_length
        for i in range(self.num_cities):
            current_city = ant.tour[i]
            next_city = ant.tour[(i+1) % self.num_cities]
            self.pheromone[current_city][next_city] +=
                pheromone_deposit
            self.pheromone[next_city][current_city] +=
                pheromone_deposit

def run(self):
    for iteration in range(self.num_iterations):
        ants = [Ant(self.num_cities, self) for i in range
        (self.num_ants)]
        for ant in ants:
            ant.construct_solution()
        self.update_pheromone(ants)
        for ant in ants:
            if ant.tour_length < self.best_tour_length:
                self.best_tour_length = ant.tour_length
                self.best_tour = ant.tour
        print(f"Iteration {iteration+1}/{self.num_
iterations}: Best Tour length = {self.best_tour_length}")
    return self.best_tour, self.best_tour_length

class Ant:
    def __init__(self, num_cities, aco_tsp):
        self.aco_tsp = aco_tsp
        self.tour = []
```

```
        self.tour = [length = 0.0
    def construct_solution(self):
        start_city = random.randint(0, self.num_cities-1)
        self.tour = [start_city]
        self.tour_length = 0.0
        visited_cities = set([self.tour])
        current_city = start_city
        while len(self.tour) < self.num_cities:
            next_city = self.aco_dsp.select_next_city(current_city, visited_cities)
            self.tour.append(next_city)
            visited_cities.add(next_city)
            self.tour_length += self.aco_dsp.cities[current_city].distance(self.aco_dsp.cities[next_city])
            current_city = next_city
        self.tour_length += self.aco_dsp.cities[self.tour[-1]].distance(self.aco_dsp.cities[self.tour[0]])

if __name__ == "__main__":
    cities = [City(0,0), City(1,3), City(4,3), City(6,1)]
    aco = ACO_TSP(cities, num_ants=10, num_iterations=100,
    alpha=1.0, beta=2.0, rho=0.5, q0=0.9)
    best_tour, best_tour_length = aco.run()

o/p :

Best tour : [2, 0, 1, 2, 3]
Best tour length : 15.1629...
```

Lab-4

Manjal
Date 21/11/24
Page 13

```
Cuckoo Search (CS):
import numpy as np
from scipy.special import gamma

def objective(x):
    return np.sum(x**2)

def levy_flight(beta, dim):
    sigma = (gamma(1+beta) * np.sin(np.pi*beta/2) /
             (gamma((1+beta)/2) * beta * np.power(2, (beta-1)/2)))**(1/beta)
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    return u/np.abs(v)**(1/beta)

def cuckoo_search(obj_func, dim, bounds, N=20, pa=0.25,
    max_iter=100):
    nests = np.random.uniform(bounds[0], bounds[1],
    (N, dim))
    fitness = np.array([obj_fun(nest) for nest in
nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for _ in range(max_iter):
        new_nests = nests.copy()
        for i in range(N):
            step = levy_flight(1.5, dim)
            new_nests[i] = nests[i] + 0.01 * step
            new_nests[i] = np.clip(new_nests[i], bounds[0],
bounds[1])
new_fitness = np.array([obj_fun(nest) for nest in
                                    new_nests])
```

Manjal
Date
Page 10

```
for i in range(n):
    if np.random.rand() < pa and new_fitness[i] <
fitness[i]:
        nests[i] = new_nests[i]
        fitness[i] = new_fitness[i]

best = nest_idx = np.argmin(fitness)
best_nest = nests[best_nest_idx]
best_fitness = fitness[best_nest_idx]
return best_nest, best_fitness
dim = 10
bounds = [-5, 5]
best_nest, best_fitness = cuckoo_search(objective, dim,
                                        bounds)
```

o/p

Best. nest [2.519, 0.8100, 2.396, 1.188, -1.860]
Best. fitness: 32.124400

Lab-5

Mangal
Date 28.11.24
Page 15

Grey wolf optimization (GWO)

```
import numpy as np
def gwo(obj_function, dim, search_agents, max_iter, lb, ub
    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)
    Alpha_score = float("inf")
    Beta_score = float("inf")
    Delta_score = float("inf")
    positions = np.random.uniform(lb, ub, (search_agents, dim))
    for iteration in range(max_iter):
        for i in range(search_agents):
            positions[i] = np.clip(positions[i], lb, ub)
            fitness = obj_function(positions[i])
            if fitness < Alpha_score:
                Alpha_score, Alpha_pos = fitness, positions[i].copy
            elif fitness < Beta_score:
                Beta_score, Beta_pos = fitness, positions[i].copy
            elif fitness < Delta_score:
                Delta_score, Delta_pos = fitness, positions[i].copy()
    print(f"Iteration {iteration+1}/{max_iter}, Best posi
{Alpha_score:.6f}")
    a = 2 - iteration * (2 / max_iter)
    for i in range(search_agents):
```

```
for j in range(dim):
    r1, r2 = np.random.rand(), np.random.rand()
    A1, C1 = 2*a*r1 - a, 2*r2
    D_alpha = abs(C1 * Alpha_pos[j] - positions[i,j])
    X1 = Alpha_pos[j] - A1 * D_alpha

    r1, r2 = np.random.rand(), np.random.rand()
    A2, C2 = 2*a*r1 - a, 2*r2
    D_beta = abs(C2 * Beta_pos[j] - positions[i,j])
    X2 = Beta_pos[j] - A2 * D_beta

    r1, r2 = np.random.rand(), np.random.rand()
    A3, C3 = 2*a*r1 - a, 2*r2
    D_delta = abs(C3 * Delta_pos[j] - positions[i,j])
    X3 = Delta_pos[j] - A3 * D_delta

    positions[i,j] = (X1 + X2 + X3)/3
return Alpha_pos, Alpha_score

def sphere_function(x):
    return np.sum(x*x)

dim = 5
Search_agents = 30
max_iter = 50
lb, ub = -10, 10

best_position, best_score = gwo(sphere_function, dim,
        Search_agents, max_iter, lb, ub)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

o/p

Best solution : [-3.1825e-05, 3.4556e-05, 3.1258e-05, 3.3126e-05]

Best Score : 6.0726e-09

Parallel cellular Algorithms and programs.

```
import numpy as np

def optimization_function(position):
    return position[0]**2 + position[1]**2

def initialize_parameters():
    grid_size = (10,10)
    num_iterations = 100
    neighbourhood_size = 1
    return grid_size, num_iterations, neighbo

def initialize_population(grid_size):
    population = np.random.uniform(-10, 10(grid_size[0],
grid_size[1], 2))
    return population

def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0], popula
.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i,j] = optimization_function
                (population[i,j])
    return fitness

def update_state(population, fitness, neighbourhood_si
    updated_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[0]):
            x_min = max(i- neighbourhood_size:0)
```

```
y_max = min(j + neighborhood_size + 1, population.shape[1])

best_neighbor = population[i, j]
best_fitness = fitness[i, j]
for x in range(x_min, x_max):
    for y in range(y_min, y_max):
        if fitness[x, y] < best_fitness:
            best_neighbor = population[x, y]
            best_fitness = fitness[x, y]

updated_population[i, j] = (population[i, j] + best_neighbor) / 2

return updated_population

def parallel_cellular_algorithm():
    grid_size, num_iterations, neighborhood_size = initialize_parameters()
    population = initialize_population(grid_size)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        fitness = evaluate_fitness(population)
        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(np.argmin(fitness), fitness.shape)]
        population = update_state(population, fitness, neighborhood_size)
```

```
    print(f"Best Solution: {best_solution}, Best
Fitness: {best_fitness}").
    return best_solution, best_fitness

if __name__ == "__main__":
    parallel_cellular_algorithm()
```

Output:

Best Solution: [-8.02149751 -7.04216626].
Best Fitness: -30.127327540600446.

Manual
Date
Page 21

Optimization via Gene Expression Algorithm

```
import numpy as np

def optimization_function(solution):
    return solution[0]**2 + solution[1]**2

def initidize_parameters():
    population_size = 50
    num_genes = 2
    mutation_rate = 0.1
    Crossover_rate = 0.8
    num_generations = 100
    return population_size, num_genes, mutation_rate,
Crossover_rate, num_generations

def initidize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_
size, num_genes))

def Evaluate_fitness(populations):
    return np.array([optimization_function(ind)
for ind in population])

def select_parents(population, fitness):
    probabilities = 1 / (fitness + 1e-6)
    probabilities /= probabilities.sum()
    indices = np.random.choice(len(population), size=len
(population), p=probabilities)
    return population[indices]
```

```
def crossover(parents, crossover_rate):
    offspring = []
    for i in range(0, len(parents), 2):
        if i+1 < len(parents) and np.random.rand() < crossover_rate:
            point = np.random.randint(1, parents.shape[1])
            offspring1 = np.concatenate((parents[i, :point], parents[i+1, point:]))
            offspring2 = np.concatenate((parents[i+1, :point], parents[i, point:]))
            offspring.extend([offspring1, offspring2])
        else:
            offspring.extend([parents[i], parents[i+1]] if i+1 < len(parents) else [parents[i]])
    return np.array(offspring)

def mutate(offspring, mutation_rate):
    for individual in offspring:
        if np.random.rand() < mutation_rate:
            gene = np.random.randint(individual.size)
            individual[gene] += np.random.normal()
    return offspring

def genetic_expression_algorithm():
    population_size, num_genes, mutation_rate, initialize_parameters()
    population = initialize_population(population_size, num_genes)
    best_solution = None
    best_fitness = float('inf')
    for generation in range(num_generations):
```

```
fitness = calculate_fitness (population)
min_fitness_idx = np.argmin (fitness)
if fitness [min_fitness_idx] < best_fitness:
    best_fitness = fitness [min_fitness_idx]
    best_solution = population [min_fitness_idx],

    parents = select_parents (population, fitness)
    offspring = crossover (parents, crossover_rate)
    population = mutate (offspring, mutation_rate)
    population = gene_expression (population)
    print (f" Generation {generation+1}: Best
Fitness = {best_fitness}")

    print (" Best Solution : {best_solution}, Best
Fitness : {best_fitness}")
    return best_solution, best_fitness.

if __name__ == "__main__":
    gene_expression_algorithm ()
```

Output:

Best Solution: [0.00434879 0.00143412],
Best Fitness: 2.09687113742899e-05.