1BM22CS279

INDEX

NAME: Sindhuja N STD.: 5th SEC.: E ROLL NO.: _____ SUB.: BIS

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	26/09/24	Genetic Algorithm	1	10 26/9/24
2	03/10/24	Genetic Algorithm	3	10 3/10/24
3.	24/10/24	Genetic Alg - Coodl.	5.	10
4.	07/11/24	Particle Swarm Optimization	8	10 07-11
5.	14/11/24	Ant colony optimization	11	10 13-11
6.	21/11/24	Cuckoo Search	14	14 21-11
7.	28/11/24	Grey wolf optimizer	17	10 28-11
8.	12/12/24	Parallel Cellular Alg.	20	10
9	19/12/24	Gene Expression Alg.	22.	10 19-12

26/09/24

Lab 1: Genetic Algorithm for Optimization Problem

A genetic algorithm is an optimization technique inspired by natural selection, where the fittest individuals in a population are selected to reproduce and create the next generation. GA's are particularly useful for solving complex problem where traditional methods struggle.

1. Chromosome: A chromosome represents a potential solution to the problem. It is typically encoded as a binary string.

2. Population: A population consists of multiple chromosomes (solutions). The population evolves over generations, with each generation ideally improving the quality of solution.

3. Fitness Function: It evaluates how good a solution is with respect to the optimization goal. Higher fitness value indicate better solution.

4. Selection: The best solution with higher fitness value are selected to create the next generation.

5. Crossover (Recombination): Two selected parent exchange materials and combine to create new offspring.

6. Mutation: Randomly alters parts of a solution to maintain diversity.

7. Generations & Iterations: The process repeats over many generations, gradually improving solutions.

8. Output: At the end of the process, the algorithm outputs the best solution it has found.

26/9/24

23/10/24

Date
Page 3

Lab 2:

Algorithm for genetic algorithm

Step 1. Initialize parameters: set the population size, mutation rate, crossover rate & no of generations.

Step 2. Generate Initial population: create a random population of potential solutions within the given bounds.

Step 3. Evaluate Fitness: calculate the fitness of each individual in the population by evaluating the objective function.

Step 4: Select Parents: select the fittest individuals from the population to reproduce based on their fitness

Step 5: Crossover perform crossover between the selected parents to create new offspring, with a probability equal to the crossover rate

Step 6. Mutate: apply mutation to the offspring, with a probability equal to mutation rate, to introduce new traits

Step 7. Replace Least Fit: replace the least fit individuals in the population with the new offspring

Step 8. Repeat: repeat steps 3-7 for a fixed number of generations or until convergence criteria are met.

Step 9. Output Best Soln: return the best soln found during the generations, which is the individual with the highest fitness.

⇒ The basic idea behind a genetic algorithm is to simulate the process of evolution by iteratively selecting, breeding and mutating a population of candidate solution.

⇒ The genetic algorithm is used for
- Optimizing problems such as scheduling, resource allocation and portfolio optimization
- machine learning to optimize parameters of models such as neural network and decision trees.
- engineering design: to optimize complex systems such as electronic circuits etc.
- computer network: to optimize network routing, scheduling and resource allocation.

⇒ Optimizing Techniques:
- selection: selecting the fittest individuals from the population to reproduce.
- crossover: combining the genetic information of two parents to create a new offspring.
- mutation: randomly changing the genetic information of an individual to introduce new traits.
- elitism: preserving the best solution from the previous generation to ensure that the best solution are not lost.
- Simulated binary crossover: A crossover technique that simulates the process of binary crossover to create new offspring.

3/10/24

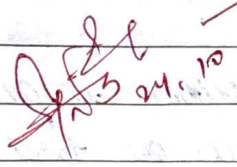
24/10/24

Lab 3: Code implementation for Genetics algo.

```
import random
import numpy as np.

def objective_function(x):
    return x**2 + 2*x + 1

def generate_initial_population(population_size, bounds):
    population = []
    for _ in range(population_size):
        x = random.uniform(bounds[0], bounds[1])
        population.append(x)
    return population.

def evaluate_fitness(population):
    fitness = []
    for x in population:
        fitness.append(objective_function(x))
    return fitness

def selection(population, fitness, num_parents):
    parents = []
    for _ in range(num_parents):
        max_fitness_idx = np.argmax(fitness)
        parents.append(population[max_fitness_idx])
        fitness[max_fitness_idx] = -float('inf')
    return parents

def crossover(parents, crossover_rate):
    offspring = []
    for _ in range(len(parents) // 2):
        parent1, parent2 = random.sample(parents, 2)
        if random.random() < crossover_rate:
```

```
    x1, x2 = parent1, parent2
    x_offspring = (x1 + x2)/2
    offspring.append(x_offspring)
return offspring

def mutation(offspring, mutation_rate, bounds):
    for i in range(len(offspring)):
        if random.random() < mutation_rate:
            offspring[i] += random.uniform(-0.1, 0.1) * (
                                bounds[1] - bounds[0])
            offspring[i] = max(bounds[0], min(offspring[i],
                                bounds[1]))
    return offspring

def genetic_algorithm(population_size, mutation_rate, crossover_
                                rate, num_gen, bounds):
    population = generate_initial_population(pop_size, bounds)
    for generation in range(num_generations):
        fitness = evaluate_fitness(population)
        parents = selection(population, fitness, population_size)
        offspring = crossover(parents, crossover_rate)
        offspring = mutation(offspring, mutation_rate, bounds)
        population = offspring + parents
    best_solution = min(population, key=objective_function)
    return best_solution

population_size = 100
mutation_rate = 0.01
crossover_rate = 0.5
num_generations = 50
bounds = (-10, 10)

best_sol = geneticalg(pop_size, mutation_rate, crossover_rate, num_
print("Best Solution:", best_solution)
```

Soln: Best Solution: 10.0



04/11/24

Lab : Particle Swarm Optimization.

→ PSO is an optimization algorithm inspired by the social behaviour of animals such as bird flocking or fish schooling. The goal is to find optimal or near-optimal solutions to complex problems by having many "particles" explore the solution space. Each particle adjusts its position based on its own experience and the experience of its neighbors, ultimately converging towards the best solution

→ ~~Algorithm~~ Code Implementation

```
import numpy as np

def rastrigin_function(x):
    A = 10
    return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i)
                            for x_i in x)

class Particle:
    def __init__(self, dim):
        self.position = np.random.uniform(-5.12, 5.12, dim)
        self.velocity = np.random.uniform(-1, 1, dim)
        self.best_position = np.copy(self.position)
        self.best_value = rastrigin_function(self.position)

class PSO:
    def __init__(self, num_particles, dim, inertia_weight, cognitive_coeff,
                 social_coeff, max_iter):
        self.num_particles = num_particles
        self.dim = dim
        self.inertia_weight = inertia_weight
```

```
self.cognitive_coeff = cognitive_coeff
self.social_coeff = social_coeff
self.max_iter = max_iter
self.swarm = [Particle(dim) for _ in range(num_particles)]
self.global_best_position = self.swarm[0].best_position
self.global_best_value = self.swarm[0].best_value

def optimize(self):
    for iteration in range(self.max_iter):
        for particle in self.swarm:
            fitness_value = rastrigin_function(particle.position)
            if fitness_value < particle.best_value:
                particle.best_value = fitness_value
                particle.best_position = np.copy(particle.position)

            if fitness_value < self.global_best_value:
                self.global_best_value = fitness_value
                self.global_best_position = np.copy(particle.best_position)

        for particle in self.swarm:
            r1 = np.random.rand(self.dim)
            r2 = np.random.rand(self.dim)

            particle.velocity = (self.inertia_weight * particle.velocity +
                                 self.cognitive_coeff * r1 * (particle.best_position -
                                 particle.position) +
                                 self.social_coeff * r2 * (self.global_best_pos))

            particle.position += particle.velocity
        return self.global_best_position, self.global_best_value
```

num_particles = 30
dim = 2
inertia_weight = 0.7
cognitive_coeff = 1.5
social_coeff = 1.5
max_iter = 100

pso = PSO(num_particles, dim, inertia_weight, cognitive_coeff,
social_coeff, max_iter)
best_position, best_value = pso.optimize()

print("Best Position:", best_position)
print("Best Value:", best_value)

Output:

Best Position: [3.13155e-10 4.9656e-09]
Best Value: 4.902744876 2e-13.

04.11

4/11/24

Lab:5 Ant Colony optimization

Ant colony optimization provides a biologically inspired approach to solving the Travelling Salesman problem by mimicking the natural behaviour of ants. By using pheromone trails and probabilistic path selection, ACO can effectively explore the solution space and converge towards optimal or near-optimal solutions for TSP.

```
import numpy as np
import random

class AntColony:
    def __init__(self, cities, num_ants, alpha, beta, rho,
                 iterations):
        self.cities = cities
        self.num_cities = len(cities)
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.best_route = None
        self.best_distance = float('inf')

    def calculate_distance_matrix(self):
        distance_matrix = np.zeros((self.num_cities, self.num_cities))
        for i in range(self.num_cities):
            for j in range(self.num_cities):
                distance_matrix[i][j] = np.linalg.norm(np.array(
                    self.cities[i]) - np.array(self.cities[j]))
        return distance_matrix
```

```
def choose_next_city(self, current_city, visited):
    probabilities = []
    for city in range(self.num_cities):
        if city not in visited:
            pheromone = self.pheromone[current_city][city] **
            heuristic = (1/self.distance_matrix[current_city][city]) **
            probabilities.append(pheromone * heuristic)
        else:
            probabilities.append(0)
    probabilities = np.array(probabilities)
    probabilities /= probabilities.sum()
    return np.random.choice(range(self.num_cities), p=
                            probabilities)

def construct_solution(self):
    for i in range(self.num_ants):
        visited = [0]
        for _ in range(1, self.num_cities):
            next_city = self.choose_next_city(visited[-1], visited)
            visited.append(next_city)
        visited.append(0)
        distance = self.calculate_route_distance(visited)
        if distance < self.best_distance:
            self.best_distance = distance
            self.best_route = visited

def calculate_route_distance(self, route):
    return sum(self.distance_matrix[route[i]][route[i+1]]
               for i in range(len(route)-1))

def update_pheromones(self):
    self.pheromone *= (1 - self.rho)
    for city in range(len(self.best_route)-1):
        self.pheromone[self.best_route[city]][self.best_route
                       [city+1]] += 1/self.best_distance
```

```
def optimize(self):
    for _ in range(self.iterations):
        self.construct_solutions()
        self.update_pheromone()
    return self.best_route, self.best_distance

if __name__ == "__main__":
    cities = [(0, 0), (1, 2), (2, 4), (3, 1), (4, 3)]
    num_ants = 10
    alpha = 1.0
    beta = 2.0
    rho = 0.5
    iterations = 10
    aco = AntColony(cities, num_ants, alpha, beta, rho, iterations)
    best_route, best_distance = aco.optimize()
    print("Best Route:", best_route)
    print("Best Distance:", best_distance)
```

Output:

Best Route: [0, 1, 2, 4, 3, 0]
Best Distance: 10.106549 70

A3.11

21-11-24

Lab 6: Cuckoo Search.

Cuckoo search is a nature inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behaviour involves laying eggs in the nest of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima.

```
import numpy as np
import random
from scipy.special import gamma

def levy_flight(beta=1.5, dim=1):
    sigma_u = np.power(gamma(1+beta) * np.sin(np.pi *
        beta/2) / gamma((1+beta)/2) * beta * np.power(np.
        pi * beta/2) ** 2), 1/beta)
    u = np.random.normal(0, sigma_u, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = u / np.power(np.abs(v), 1/beta)
    return step

def initialize_population(n_nests, n_dim, lower_bound,
                          upper_bound):
    return np.random.uniform(lower_bound, upper_bound,
                             (n_nests, n_dim))

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

def cuckoo_search(n_nests, n_dim, lower_bound, upper
                  bound, max_iter, pa=0.25):
    nest = initialize_population(n_nests, n_dim, lower_bound,
                                 upper_bound)
```

```
fitness = np.array([fitness_function(nest) for nest in
                                        nests])
best_idx = np.argmin(fitness)
best_nest = nests[best_idx]
best_fitness = fitness[best_idx]

for iteration in range(max_iter):
    for i in range(n_nests):
        step = levy_flight(d, n_dim)
        new_nest = nests[i] + step * (nests[i] - best_nest)
        new_nest = np.clip(new_nest, lower_bound,
                                        upper_bound)
        new_fitness = fitness_function(new_nest)
        if new_fitness < fitness[i]:
            nests[i] = new_nest
            fitness[i] = new_fitness
            if new_fitness < best_fitness:
                best_nest = new_nest
                best_fitness = new_fitness

    for i in range(n_nests):
        if random.random() < pa:
            nests[i] = np.random.uniform(lower_bound,
                                upper_bound, n_dim)
            fitness[i] = fitness_function(nests[i])

    if (iteration+1) % 100 == 0 or iteration == max_iter-1:
        Print(f"Iteration {iteration+1}: Best Fitness: {best_fitness}
return best_nest, best_fitness.

n_nests = 25
n_dim = 10
lower_bound = -5
upper_bound = 5.
```

max_itr = 1000
pa = 0.25.

best_solution, best_solution_fitness = cuckoo_search(
n_nests, n_dim, lower_bound, upper_bound,
max_itr, pa)

print("Best solution found:", best_solution)
print("Best fitness value:", best_solution_fitness)

Output:

Iteration 100, Best Fitness: 6.398706003611
Iteration 200, -"- : 6.398706003611
Iteration 300, -"- : 6.398706003611
Iteration 400, -"- : 6.398706003611
Iteration 500, -"- : 5.880452968742452
Iteration 600, -"- : 4.591078603855321
Iteration 700, -"- : 3.793057326065373
Iteration 800, -"- : 3.793057326065373
Iteration 900, -"- : 3.793057326065373
Iteration 1000, -"- : 3.793057326065373

Best Solution found: [0.1980991 1.0031885 -0.2665895
-0.3667659 -0.2855916 1.0051995 -0.5890082
-0.3542278 -0.961117 0.3904871]
Best fitness value: 3.793057326065373.

21.11.24

3/11/24

classmate
Date
Page 17

Lab 7: Grey wolf Optimizer

The grey wolf optimizer is a swarm intelligent algo inspired by the social hierarchy and hunting behaviour of grey wolves. It mimics the leadership structure of alpha, beta, & omega wolves and their collaborative hunting strategies

```
import numpy as np

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

def grey_wolf_optimizer(obj_func, dim, lb, ub, wolves_count, iterations):
    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")

    wolves_positions = np.random.uniform(lb, ub, (wolves_count, dim))

    for iteration in range(iterations):
        for i in range(wolves_count):
            fitness = obj_func(wolves_positions[i])
            if fitness < alpha_score:
                alpha_score, alpha_pos = fitness, wolves_positions[i].copy()
            elif fitness < beta_score:
                beta_score, beta_pos = fitness, wolves_positions[i].copy()
```

elif fitness < delta_scor:
delta_scor, delta_pos = fitness, wolves_position[i]
copy

for i in range(wolves_count):
for j in range(dim):
a = 2 - 2 * (iteration / iterations)
r1, r2 = np.random.rand(), np.random.rand()

A1 = 2*a*r1 - a
C1 = 2*r2

D_alpha = abs(C1 * alpha_pos[j] - wolves_position[i][j])
X1 = alpha_pos[j] - A1 * D_alpha
r1, r2 = np.random.rand(), np.random.rand()
A2 = 2*a*r1 - a
C2 = 2*r2

D_beta = abs(C2 * beta_pos[j] - wolves_positions[i][j])
X2 = beta_pos[j] - A2 * D_beta

r1, r2 = np.random.rand(), np.random.rand()
A3 = 2*a*r1 - a
C3 = 2*r2

D_delta = abs(C3 * delta_pos[j] - wolves_positions[i][j])
X3 = delta_pos[j] - A3 * D_delta

wolves_positions[i][j] = (X1 + X2 + X3) / 3

wolves_positions[i] = np.clip(wolves_positions[i], lb, ub)

return alpha_pos, alpha_scor

dimension = 5
lower_bound = -10
upper_bound = 10
wolves = 30
max_iteration = 50

best_position, best_score = grey_wolf_optimizer(
objective_function, dimension, lower_bound,
upper_bound, wolves, maxiteration)

print("Best position", best_position)
print("Best score:", best_score)

Output

Best position: [6.39416661 6.40050565 -6.59530749
7.98219731 -7.16874410e-06.]
Best score: 0.40456295340.

28.11.

19/12/24

Lab 8: Parallel Cellular Algorithm.

Parallel cellular algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner.

```
import numpy as np.
def rastrigin_function(x):
    A=10
    return A * len(x) + sum([(xi ** 2 - A * np.cos(2 * np.pi * xi)
                                                    for xi in x])

class Parallel Cellular Algorithm:
    def __init__(self, grid_size=(5,5), dim=2, neighborhood
                                        'moore', iterations=100):
        self.grid_size = grid_size
        self.dim = dim
        self.neighborhood = neighborhood
        self.bounds = bounds
        self.grid = np.random.uniform(bounds[0], bounds[1],
                                        grid_size[1], dim))
        self.fitness = np.zeros((grid_size[0], grid_size[1]))

    def evaluate_fitness(self):
        for i in range(self.grid_size[0]):
            for j in range(self.grid_size[1]):
                self.fitness[i,j] = rastrigin_fnc(self.grid[i,j])

    def get_neighbors(self, i, j):
        neighbors = []
        directions = []
        if self.neighborhood == 'moore':
            directions = [(-1,-1), (-1, 0), (-1, 1)]
        elif self.neighborhood == 'von_neuman':
            directions = [(-1, 0), (0, -1) (0, 1)]
```

```
def update_states(self):
    new_grid = self.grid.copy()
    for i in range(self.grid_size[0]):
        for j in range(self.grid.size[1]):
            neighbors = self.get_neighbors(i, j)
    self.grid = new_grid

def run(self):
    for iteration in range(self.iterations):
        self.evaluate_fitness()
        self.update_states()
        print(f"Iteration {iteration+1}/{self.iterations}")

    print("Optimization complete")
    print(f"Best solution: {self.best_solution}")
    print(f"Best Fitness: {self.best_fitness:.6f}")

if __name__ == "__main__":
    pca = ParallelCellularAlgorithm(grid_size=(10,10),
                                    dim=2, neighborhood='moore')
    pca.run()
```

Output.

```
Optimization Complete
Best Solution: [-0.033945   -0.034027]
Best Fitness: 0.451717
```

19/11/24

Lab 9: Optimization via Gene Expression Algorithm

Gene Expression Algo are inspired by the biological process of gene expression in living organisms. The process involves the translation of genetic information encoded in DNA into functional proteins.

```
import numpy as np
import random
def rastrigin_function(x):
    A = 10
    return A * len(x) + sum([(xi ** 2 - A * np.cos(2 * np.pi
                                * xi)) for xi in x])

class GeneExpressionAlgorithm:
    def __init__(self, pop_size=100, gene_length=10, bounds=(-5.12, 5.12),
    self.pop_size = pop_size
    self.gene_length = gene_length
    self.mutation_rate = mutation_rate
    self.bounds = bounds

def random_gene_sequence(self):
    return np.random.uniform(self.bounds[0], self.bounds[1],
                             size=(self.gene_length))

def evaluate_fitness(self, gene_sequence):
    return rastrigin_function(gene_sequence)

def selection(self):
    tournament_size = 3
    selected = []
    for _ in range(self.pop_size):
        tournament = random.sample(self.population, tournament_
                                   size)

    return selected
```

```
def mutate(self, gene_sequence):
    for i in range(len(gen_sequence)):
        if random.random() < self.mutation_rate:
            gene_sequence[i] += np.random.uniform(-1.0, 1.0)
            gene_sequence[i] = np.clip(gene_sequence[i],
    return gene_sequence.                    self.bounds[i])

def gene_expression(self, gene_sequence):
    return gene_sequence.

def run(self):
    for generation in range(self.generations):
        fitness_values = [self.evaluate_fitness(ind) for ind
                                                in self.population]
        best_idx = np.argmin(fitness_value)
        selected_pop = self.selection()
        next_population = []
    for i in range(0, self.pop_size, 2):
        parent1, select_pop[i]
        parent2 = select_pop[i+1]
        offspring1, offspring2 = self.crossover(p1, p2)
        next_pop.append(self.mutate(offspring1,))
    self.population = next_population

print("Optimization complete")
print(f"Best Sol: {self.best_sol}")
print(f"Best Fitness: {self.best_fitness:.6f}")

ga.run().
```

O/P

```
Optimization Complete
Best Sol: [-0.02796851  -0.01908597]
Best Fitness: 0.226974.
```