Bio inspired System Lab 1BM22CS158 Mohammed Shuraim

1)GENETIC ALGORITHM

```
import random

# Define the fitness function
def fitness_function(x):
    return x**2  # Example function: f(x) = x^2

# Generate initial population
def generate_population(size, x_min, x_max):
    return [random.uniform(x_min, x_max) for _ in range(size)]

# Selection process
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parents = random.choices(population, weights=selection_probs, k=2)
    return parents

# Crossover process
def crossover(parent1, parent2):
    alpha = random.random()
    child = alpha * parent1 + (1 - alpha) * parent2
    return child

# Mutation process
def mutate(child, mutation_rate, x_min, x_max):
    if random.random() < mutation_rate:
        child = random.uniform(x_min, x_max)
    return child

# Genetic Algorithm
def genetic_algorithm(pop_size, generations, mutation_rate, x_min, x_max):
    population = generate_population(pop_size, x_min, x_max)
    for generation in range(generations):
        fitnesses = [fitness_function(ind) for ind in population]
        new_population = []
        for _ in range(pop_size):
            parent1, parent2 = select_parents(population, fitnesses)
            child = crossover(parent1, parent2)
            child = mutate(child, mutation_rate, x_min, x_max)
            new_population.append(child)
        population = new_population
    best_solution = max(population, key=fitness_function)
    return best_solution

# User inputs
pop_size = int(input("Enter population size: "))
generations = int(input("Enter number of generations: "))
mutation_rate = float(input("Enter mutation rate (0-1): "))
x_min = float(input("Enter minimum value of x: "))
x_max = float(input("Enter maximum value of x: "))

# Run the genetic algorithm
best_solution = genetic_algorithm(pop_size, generations, mutation_rate, x_min, x_max)
print(f"The best solution found is: {best_solution} with fitness value: {fitness_function(best_solution)}")
```

```
Enter population size: 20
Enter number of generations: 3
Enter mutation rate (0-1): 2
Enter minimum value of x: 4
Enter maximum value of x: 8
The best solution found is: 7.871117037734696 with fitness value: 61.95448342171742
```

Double-click (or enter) to edit

2 Ant colony

```
import numpy as np
import random

class AntColony:
    def __init__(self, distance_matrix, n_ants, n_iterations, decay, alpha=1, beta=1):
        self.distance_matrix = distance_matrix
        self.pheromone = np.ones(distance_matrix.shape) / len(distance_matrix)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha  # Pheromone importance
        self.beta = beta    # Distance importance
        self.all_indices = range(len(distance_matrix))

    def run(self):
        shortest_path = None
        all_time_shortest_path = ("path", np.inf)

        for _ in range(self.n_iterations):
            all_paths = self.generate_all_paths()
            self.update_pheromones(all_paths)
            shortest_path = min(all_paths, key=lambda x: x[1])
            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path

        return all_time_shortest_path

    def generate_all_paths(self):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.generate_path(0)  # Start from city 0
            path_dist = self.calculate_path_distance(path)
            all_paths.append((path, path_dist))
        return all_paths

    def generate_path(self, start):
        path = [start]
        visited = set(path)
        while len(visited) < len(self.distance_matrix):
            move = self.select_next_city(path[-1], visited)
            path.append(move)
            visited.add(move)
        path.append(start)  # Return to starting city
        return path

    def select_next_city(self, current_city, visited):
        pheromone = np.copy(self.pheromone[current_city])
        pheromone[list(visited)] = 0  # Avoid visiting already visited cities

        probabilities = pheromone ** self.alpha * ((1 / self.distance_matrix[current_city]) ** self.beta)
        probabilities /= probabilities.sum()  # Normalize probabilities

        next_city = np.random.choice(self.all_indices, p=probabilities)
        return next_city

    def calculate_path_distance(self, path):
        total_dist = 0
        for i in range(len(path) - 1):
            total_dist += self.distance_matrix[path[i]][path[i + 1]]
        return total_dist

    def update_pheromones(self, all_paths):
        self.pheromone *= (1 - self.decay)  # Pheromone evaporation
        for path, dist in all_paths:
            for i in range(len(path) - 1):
                self.pheromone[path[i]][path[i + 1]] += 1 / dist  # Update pheromone based on path quality

# Example: A 4-city TSP problem
if __name__ == "__main__":
    distance_matrix = np.array([[np.inf, 12, 12, 15],
                                [12, np.inf, 13, 14],
                                [12, 13, np.inf, 11],
                                [15, 14, 11, np.inf]])
```

```
    colony = AntColony(distance_matrix, n_ants=10, n_iterations=100, decay=0.1, alpha=1, beta=2)
    best_path = colony.run()
    print("Best path found:", best_path)
```

```
Best path found: ([0, 1, 3, 2, 0], 49.0)
```

3 Particle Swarm Optimization

```
import numpy as np

# Objective function (Example: Rastrigin function)
def objective_function(position):
    return sum([x**2 - 10 * np.cos(2 * np.pi * x) + 10 for x in position])

# Particle Swarm Optimization
class Particle:
    def __init__(self, dimensions):
        self.position = np.random.uniform(-10, 10, dimensions)  # Initialize position
        self.velocity = np.random.uniform(-1, 1, dimensions)    # Initialize velocity
        self.best_position = self.position.copy()               # Personal best position
        self.best_score = float('inf')                          # Best score for personal best

    def update_velocity(self, global_best_position, inertia, cognitive_const, social_const):
        r1, r2 = np.random.rand(), np.random.rand()
        cognitive = cognitive_const * r1 * (self.best_position - self.position)
        social = social_const * r2 * (global_best_position - self.position)
        self.velocity = inertia * self.velocity + cognitive + social

    def update_position(self):
        self.position += self.velocity

# PSO Algorithm
def particle_swarm_optimization(objective_func, dimensions, num_particles, max_iter):
    inertia = 0.5           # Inertia weight
    cognitive_const = 1.5    # Cognitive constant
    social_const = 1.5       # Social constant

    # Initialize particles
    swarm = [Particle(dimensions) for _ in range(num_particles)]
    global_best_position = np.random.uniform(-10, 10, dimensions)
    global_best_score = float('inf')

    for iteration in range(max_iter):
        for particle in swarm:
            # Evaluate fitness
            fitness = objective_func(particle.position)
            # Update personal best
            if fitness < particle.best_score:
                particle.best_score = fitness
                particle.best_position = particle.position.copy()

            # Update global best
            if fitness < global_best_score:
                global_best_score = fitness
                global_best_position = particle.position.copy()

        # Update velocity and position for each particle
        for particle in swarm:
            particle.update_velocity(global_best_position, inertia, cognitive_const, social_const)
            particle.update_position()

        print(f"Iteration {iteration+1}/{max_iter}, Global Best Score: {global_best_score}")

    return global_best_position, global_best_score

# Example usage
best_position, best_score = particle_swarm_optimization(objective_function, dimensions=2, num_particles=30, max_iter=100)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

```
Iteration 1/100, Global Best Score: 18.36113425179184
Iteration 2/100, Global Best Score: 8.686660822967248
Iteration 3/100, Global Best Score: 8.686660822967248
Iteration 4/100, Global Best Score: 1.4949397829370117
Iteration 5/100, Global Best Score: 1.4949397829370117
Iteration 6/100, Global Best Score: 1.4949397829370117
Iteration 7/100, Global Best Score: 1.4949397829370117
Iteration 8/100, Global Best Score: 1.4949397829370117
Iteration 9/100, Global Best Score: 1.4949397829370117
Iteration 10/100, Global Best Score: 1.4949397829370117
Iteration 11/100, Global Best Score: 1.4949397829370117
Iteration 12/100, Global Best Score: 1.1319169085968799
Iteration 13/100, Global Best Score: 1.0093101190758844
Iteration 14/100, Global Best Score: 0.9999631047686908
Iteration 15/100, Global Best Score: 0.9969229153925063
Iteration 16/100, Global Best Score: 0.9967168919316318
Iteration 17/100, Global Best Score: 0.9967168919316318
Iteration 18/100, Global Best Score: 0.9964681775893869
Iteration 19/100, Global Best Score: 0.9960521853175219
Iteration 20/100, Global Best Score: 0.9960521853175219
Iteration 21/100, Global Best Score: 0.9960521853175219
Iteration 22/100, Global Best Score: 0.995574323012157
Iteration 23/100, Global Best Score: 0.9951576576478445
Iteration 24/100, Global Best Score: 0.9951576576478445
Iteration 25/100, Global Best Score: 0.9950687048708495
Iteration 26/100, Global Best Score: 0.9949851281462312
Iteration 27/100, Global Best Score: 0.9949656545514554
Iteration 28/100, Global Best Score: 0.9949631075269068
Iteration 29/100, Global Best Score: 0.994961704637765
Iteration 30/100, Global Best Score: 0.994961704637765
Iteration 31/100, Global Best Score: 0.9949609086921605
Iteration 32/100, Global Best Score: 0.9949590989710657
Iteration 33/100, Global Best Score: 0.9949590989710657
Iteration 34/100, Global Best Score: 0.9949590989710657
Iteration 35/100, Global Best Score: 0.9949590989710657
Iteration 36/100, Global Best Score: 0.9949590989710657
Iteration 37/100, Global Best Score: 0.9949590989710657
Iteration 38/100, Global Best Score: 0.9949590989710657
Iteration 39/100, Global Best Score: 0.9949590989710657
Iteration 40/100, Global Best Score: 0.9949590989710657
Iteration 41/100, Global Best Score: 0.9949590583503163
Iteration 42/100, Global Best Score: 0.9949590583503163
Iteration 43/100, Global Best Score: 0.9949590571452429
Iteration 44/100, Global Best Score: 0.9949590571452429
Iteration 45/100, Global Best Score: 0.9949590571452429
Iteration 46/100, Global Best Score: 0.9949590571215658
Iteration 47/100, Global Best Score: 0.9949590570949134
Iteration 48/100, Global Best Score: 0.9949590570949134
Iteration 49/100, Global Best Score: 0.9949590570949134
Iteration 50/100, Global Best Score: 0.9949590570949134
Iteration 51/100, Global Best Score: 0.9949590570949134
Iteration 52/100, Global Best Score: 0.9949590570949134
Iteration 53/100, Global Best Score: 0.9949590570949134
Iteration 54/100, Global Best Score: 0.9949590570949134
Iteration 55/100, Global Best Score: 0.9949590570934284
Iteration 56/100, Global Best Score: 0.9949590570934284
Iteration 57/100, Global Best Score: 0.9949590570934284
Iteration 58/100, Global Best Score: 0.9949590570934284
```

4 Cuckoo Search optimization

```python
import numpy as np

# Objective function (example: Sphere function)
def objective_function(x):
    return np.sum(x ** 2)

# Levy flight implementation
def levy_flight(Lambda, dim, alpha=1.0):
    u = np.random.normal(0, 1, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = alpha * (u / (np.abs(v) ** (1 / Lambda)))  # Lévy step
    return step

# Cuckoo Search Algorithm
def cuckoo_search(n, max_generations, pa, lower_bound, upper_bound, dim):
    # Step 1: Initialize nests randomly
    nests = np.random.uniform(lower_bound, upper_bound, size=(n, dim))
    fitness = np.array([objective_function(nest) for nest in nests])
```

```
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    # Iterative optimization
    for t in range(max_generations):
        # Rule 1: Generate new solutions via Lévy flight
        for i in range(n):
            new_nest = nests[i] + levy_flight(1.5, dim)
            new_nest = np.clip(new_nest, lower_bound, upper_bound)
            new_fitness = objective_function(new_nest)

            # Rule 2: Replace nests if better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

                # Update global best
                if new_fitness < best_fitness:
                    best_nest = new_nest
                    best_fitness = new_fitness

        # Rule 3: Abandon some nests and create new random ones
        abandon = np.random.rand(n) < pa
        nests[abandon] = np.random.uniform(lower_bound, upper_bound, size=(np.sum(abandon), dim))
        fitness[abandon] = np.array([objective_function(nest) for nest in nests[abandon]])

    return best_nest, best_fitness

# Parameters
n = 25  # Number of nests
dim = 5  # Dimensionality of the problem
max_generations = 100  # Max iterations
pa = 0.25  # Abandonment probability
lower_bound = -10  # Lower bound of the search space
upper_bound = 10  # Upper bound of the search space

# Run Cuckoo Search
best_solution, best_value = cuckoo_search(n, max_generations, pa, lower_bound, upper_bound, dim)

print("Best solution found:", best_solution)
print("Best objective value:", best_value)
```

```
Best solution found: [ 0.20497829 -0.33362922 -0.59123456 -0.74011305 -1.28606162]
Best objective value: 2.7046046892112336
```

5 Grey Wolf Optimizer

```
import numpy as np

# Objective function (e.g., Sphere function)
def objective_function(position):
    return sum(x**2 for x in position)

# Grey Wolf Optimizer
def grey_wolf_optimizer(obj_function, dim, pop_size, max_iter, bounds=(-10, 10)):
    a = 2  # Coefficient, decreases linearly from 2 to 0
    alpha_position = np.zeros(dim)
    alpha_score = float('inf')  # Best fitness (alpha)
    beta_position = np.zeros(dim)
    beta_score = float('inf')  # Second-best fitness (beta)
    delta_position = np.zeros(dim)
    delta_score = float('inf')  # Third-best fitness (delta)

    # Initialize the positions of the wolves
    wolves = np.random.uniform(bounds[0], bounds[1], (pop_size, dim))

    for iteration in range(max_iter):
        for i, wolf in enumerate(wolves):
            fitness = obj_function(wolf)

            # Update alpha, beta, and delta
            if fitness < alpha_score:
                delta_position = beta_position.copy()
```

```
                delta_score = beta_score
                beta_position = alpha_position.copy()
                beta_score = alpha_score
                alpha_position = wolf.copy()
                alpha_score = fitness
            elif fitness < beta_score:
                delta_position = beta_position.copy()
                delta_score = beta_score
                beta_position = wolf.copy()
                beta_score = fitness
            elif fitness < delta_score:
                delta_position = wolf.copy()
                delta_score = fitness

        # Update positions
        for i, wolf in enumerate(wolves):
            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_position - wolf)
            X1 = alpha_position - A1 * D_alpha

            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_position - wolf)
            X2 = beta_position - A2 * D_beta

            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_position - wolf)
            X3 = delta_position - A3 * D_delta

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

        # Linearly decrease a
        a -= 2 / max_iter

        print(f"Iteration {iteration+1}/{max_iter}, Alpha Score: {alpha_score}")

    return alpha_position, alpha_score

# Example usage
best_position, best_score = grey_wolf_optimizer(objective_function, dim=2, pop_size=30, max_iter=100)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

```
Iteration 76/100, Alpha Score: 6.438062741100612e-59
Iteration 77/100, Alpha Score: 4.527257759890036e-59
Iteration 78/100, Alpha Score: 3.3483182955898075e-59
Iteration 79/100, Alpha Score: 2.4895995086539206e-59
Iteration 80/100, Alpha Score: 2.056879985021375e-59
Iteration 81/100, Alpha Score: 1.651567471453138e-59
Iteration 82/100, Alpha Score: 1.2575869733054122e-59
Iteration 83/100, Alpha Score: 1.010966375158604e-59
Iteration 84/100, Alpha Score: 8.115265985124924e-60
Iteration 85/100, Alpha Score: 6.367483102766966e-60
Iteration 86/100, Alpha Score: 6.084544645736791e-60
Iteration 87/100, Alpha Score: 4.9652127243689097e-60
Iteration 88/100, Alpha Score: 4.576507456384005e-60
Iteration 89/100, Alpha Score: 3.8964925219752986e-60
Iteration 90/100, Alpha Score: 3.580478177703101e-60
Iteration 91/100, Alpha Score: 3.102988105420075e-60
Iteration 92/100, Alpha Score: 2.936828514858608e-60
Iteration 93/100, Alpha Score: 2.671008236898743e-60
Iteration 94/100, Alpha Score: 2.4811269749912955e-60
Iteration 95/100, Alpha Score: 2.3383305537762118e-60
Iteration 96/100, Alpha Score: 2.206454382871867e-60
Iteration 97/100, Alpha Score: 2.1121148046984019e-60
Iteration 98/100, Alpha Score: 2.0185177719072882e-60
Iteration 99/100, Alpha Score: 1.9403778098441208e-60
Iteration 100/100, Alpha Score: 1.9173501698915915e-60
Best Position: [9.19241999e-31 1.03554059e-30]
Best Score: 1.9173501698915915e-60
```

6 parallel cellular optimization

```
import numpy as np

# Define the objective function
def objective_function(x):
    return x**2 - 4*x + 4

# Initialize the grid
def initialize_grid(grid_size, search_range):
    return np.random.uniform(search_range[0], search_range[1], (grid_size, grid_size))

# Compute fitness for the grid
def evaluate_fitness(grid, objective_function):
    return objective_function(grid)

# Update the grid based on neighborhood average
def update_grid(grid):
    new_grid = np.copy(grid)
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            # Get neighbors' values
            neighbors = []
            for di in [-1, 0, 1]:
                for dj in [-1, 0, 1]:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
                        neighbors.append(grid[ni, nj])
            # Update state to the average of neighbors
            new_grid[i, j] = np.mean(neighbors)
    return new_grid

# Main function to run the algorithm
def parallel_cellular_algorithm(grid_size, search_range, iterations):
    grid = initialize_grid(grid_size, search_range)  # Step 2: Initialize grid
    for _ in range(iterations):
        fitness = evaluate_fitness(grid, objective_function)  # Step 3: Evaluate fitness
        grid = update_grid(grid)  # Step 4: Update states
    # Find the best solution
    best_value = grid[np.unravel_index(np.argmin(fitness), fitness.shape)]
    return best_value, objective_function(best_value)

# Parameters
grid_size = 10  # 10x10 grid
search_range = (-10, 10)  # Search range for cell values
iterations = 100  # Number of iterations

# Run the algorithm
best_value, best_fitness = parallel_cellular_algorithm(grid_size, search_range, iterations)
```

```
# Output the results
print(f"Best Value: {best_value}")
print(f"Best Fitness: {best_fitness}")
```

```
Best Value: -0.7769646689183809
Best Fitness: 7.711532772420973
```

7 gene expression

```
import numpy as np
import random

# Define the Rastrigin function (objective function)
def rastrigin_function(x):
    A = 10
    n = len(x)
    return A * n + sum([(xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Initialize population
def initialize_population(pop_size, gene_length, search_range):
    return [np.random.uniform(search_range[0], search_range[1], gene_length) for _ in range(pop_size)]

# Evaluate fitness
def evaluate_fitness(population):
    return [rastrigin_function(ind) for ind in population]

# Selection (tournament selection)
def selection(population, fitness):
    selected = []
    for _ in range(len(population)):
        i, j = random.sample(range(len(population)), 2)
        selected.append(population[i] if fitness[i] < fitness[j] else population[j])
    return selected

# Crossover (uniform crossover)
def crossover(parent1, parent2):
    child = []
    for p1, p2 in zip(parent1, parent2):
        child.append(p1 if random.random() < 0.5 else p2)
    return np.array(child)

# Mutation (random mutation)
def mutate(individual, mutation_rate, search_range):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.uniform(search_range[0], search_range[1])
    return individual

# Gene Expression Algorithm
def gene_expression_algorithm(pop_size, gene_length, generations, mutation_rate, search_range):
    # Step 1: Initialize population
    population = initialize_population(pop_size, gene_length, search_range)

    for generation in range(generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Step 3: Selection
        selected_population = selection(population, fitness)

        # Step 4: Crossover and Mutation
        next_population = []
        for i in range(0, len(selected_population), 2):
            if i + 1 < len(selected_population):
                # Crossover
                child1 = crossover(selected_population[i], selected_population[i + 1])
                child2 = crossover(selected_population[i + 1], selected_population[i])
            else:
                child1 = selected_population[i]
                child2 = selected_population[i]
            # Mutation
            next_population.append(mutate(child1, mutation_rate, search_range))
            next_population.append(mutate(child2, mutation_rate, search_range))
```

```
            next_population.append(mutate(child2, mutation_rate, search_range))

        population = next_population[:pop_size]  # Maintain population size

    # Final fitness evaluation
    fitness = evaluate_fitness(population)
    best_individual = population[np.argmin(fitness)]
    return best_individual, rastrigin_function(best_individual)

# Parameters
pop_size = 50
gene_length = 10
generations = 100
mutation_rate = 0.1
search_range = (-5.12, 5.12)  # Search range for Rastrigin function

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm(pop_size, gene_length, generations, mutation_rate, search_range)

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

```
Best Solution: [-1.01547995  0.07075383 -0.02144954 -1.02279118  1.05152707 -2.00221714
 -0.28127705 -1.02468392 -0.95537354 -0.99059453]
Best Fitness: 24.43384866081925
```