/lo

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shilpa K M (1BM23CS419)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Shilpa K M (1BM23CS419),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Saritha
Assistant Professor
Department of CSE, BMSCE

Index

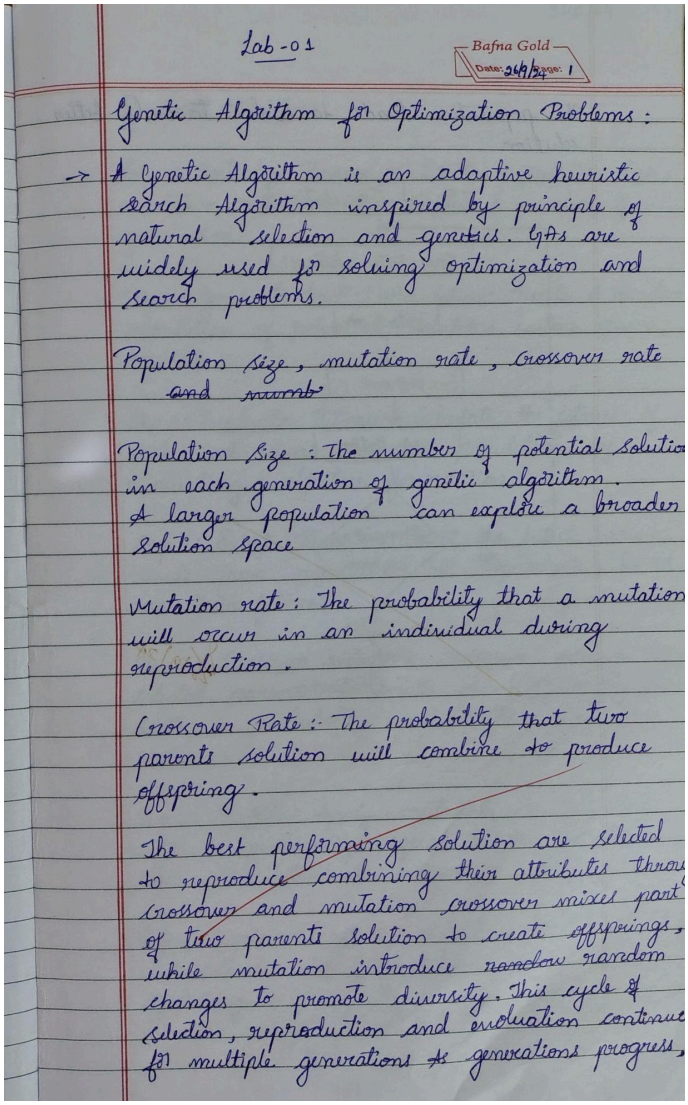
Github Link:
https://github.com/Sakshiibr/BIS_LAB

Program 1

Genetic Algorithm

A Genetic Algorithm (GA) is an optimization technique based on the concept of natural evolution. It works by representing potential solutions as individuals in a population. The algorithm improves these solutions over time using processes like selection (choosing the best solutions), crossover (combining traits), and mutation (introducing small changes), ultimately evolving towards the best possible solution.

Algorithm:


Lab-01

Bafna Gold
Date: 24/1/24 Page: 1

Genetic Algorithm for Optimization Problems:

→ A Genetic Algorithm is an adaptive heuristic search Algorithm inspired by principle of natural selection and genetics. GAs are widely used for solving optimization and search problems.

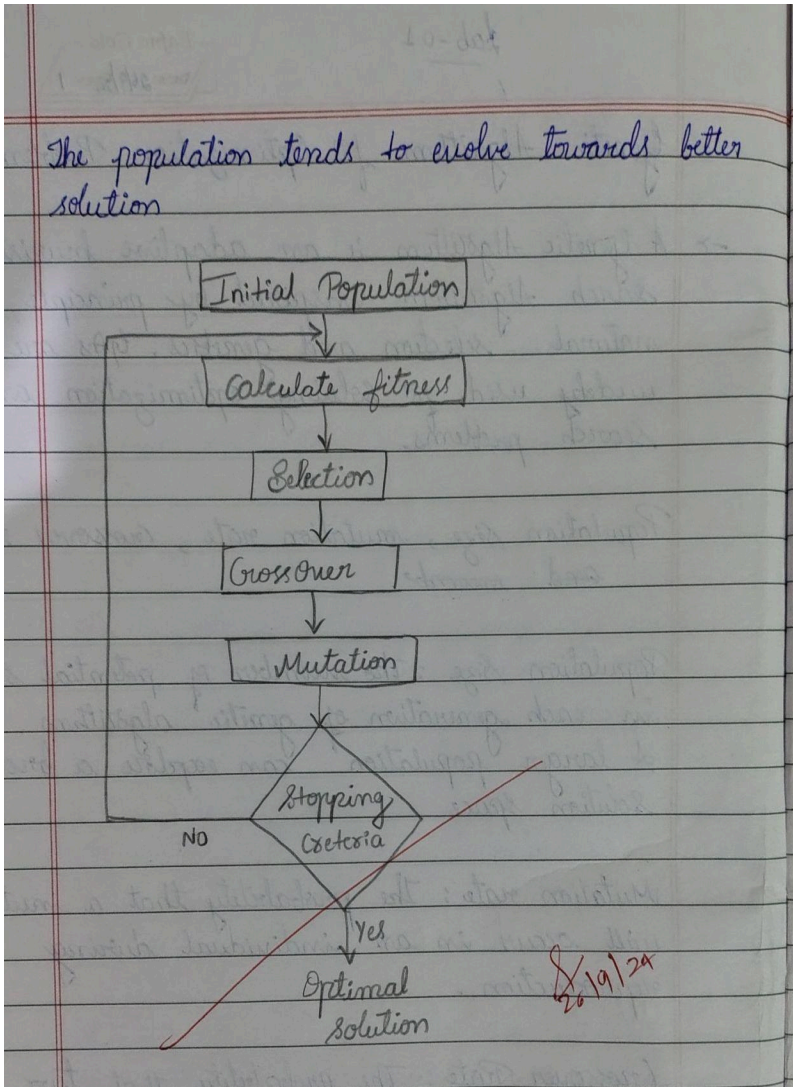
Population size, mutation rate, crossover rate and numb

Population Size: The number of potential solution in each generation of genetic algorithm. A larger population can explore a broader solution space

Mutation rate: The probability that a mutation will occur in an individual during reproduction.

Crossover Rate: The probability that two parents solution will combine to produce offspring.

The best performing solution are selected to reproduce combining their attributes throu Crossover and mutation crossover mixes part of two parents solution to create offsprings, while mutation introduce ~~random~~ random changes to promote diversity. This cycle of selection, reproduction and evaluation continue for multiple generations as generations progress,



Code:

```
import numpy as np
import random
import matplotlib.pyplot as plt

# Define the problem (fitness function)
def fitness_function(x):
    return x * np.sin(10 * np.pi * x) + 1.0

# Parameters
population_size = 20
mutation_rate = 0.1
crossover_rate = 0.7
generations = 50
bounds = [0, 1]  # Range of x values
```

```
# Create the initial population
def create_population(size, bounds):
    return np.random.uniform(bounds[0], bounds[1], size)

# Evaluate fitness of the population
def evaluate_fitness(population):
    return np.array([fitness_function(ind) for ind in population])

# Select parents using roulette wheel selection
def select_parents(population, fitness):
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    indices = np.random.choice(len(population), size=2, p=probabilities)
    return population[indices]

# Perform crossover
def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        point = np.random.uniform()
        child1 = point * parent1 + (1 - point) * parent2
        child2 = (1 - point) * parent1 + point * parent2
        return child1, child2
    return parent1, parent2

# Perform mutation
def mutate(individual, bounds):
    if np.random.rand() < mutation_rate:
        mutation = np.random.uniform(bounds[0], bounds[1])
        return mutation
    return individual

# Genetic Algorithm Implementation
def genetic_algorithm():
    population = create_population(population_size, bounds)
    best_fitness_values = []

    for generation in range(generations):
        fitness = evaluate_fitness(population)
        new_population = []

        # Save the best fitness of the current generation
        best_fitness_values.append(np.max(fitness))

        for _ in range(population_size // 2):  # Produce new generation
            parent1, parent2 = select_parents(population, fitness)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, bounds)
            child2 = mutate(child2, bounds)
            new_population.extend([child1, child2])
```

```
        # Ensure individuals are within bounds
        population = np.clip(new_population, bounds[0], bounds[1])

    # Return the best solution found and fitness over generations
    best_individual = population[np.argmax(fitness)]
    return best_individual, best_fitness_values

# Run the Genetic Algorithm
best_solution, fitness_history = genetic_algorithm()

# Print the results
print("Best solution found:", best_solution)
print("Fitness of the best solution:", fitness_function(best_solution))
```

Output:

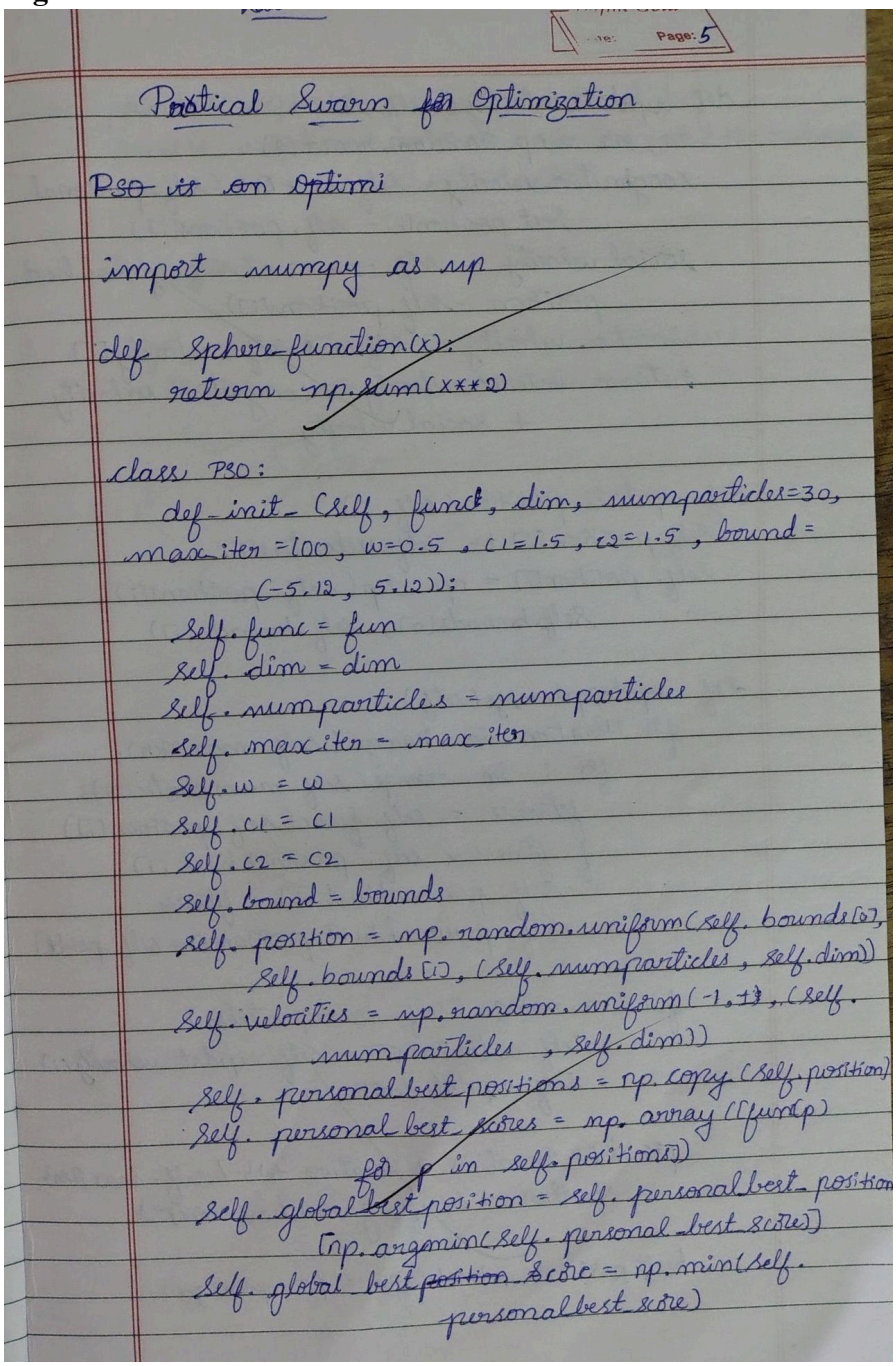
```
Best solution found: 0.6558044164581247
Fitness of the best solution: 1.644931214483274
```

Program 2

Particle Swarm Optimisation for function Optimisation

Particle Swarm Optimization (PSO) is an optimization algorithm inspired by the way birds flock or fish school. It uses a group of particles, each representing a potential solution, that move through the search space collectively. The particles adjust their positions based on their own experience and the best positions found by the swarm, working together to find the optimal solution.

Algorithm:



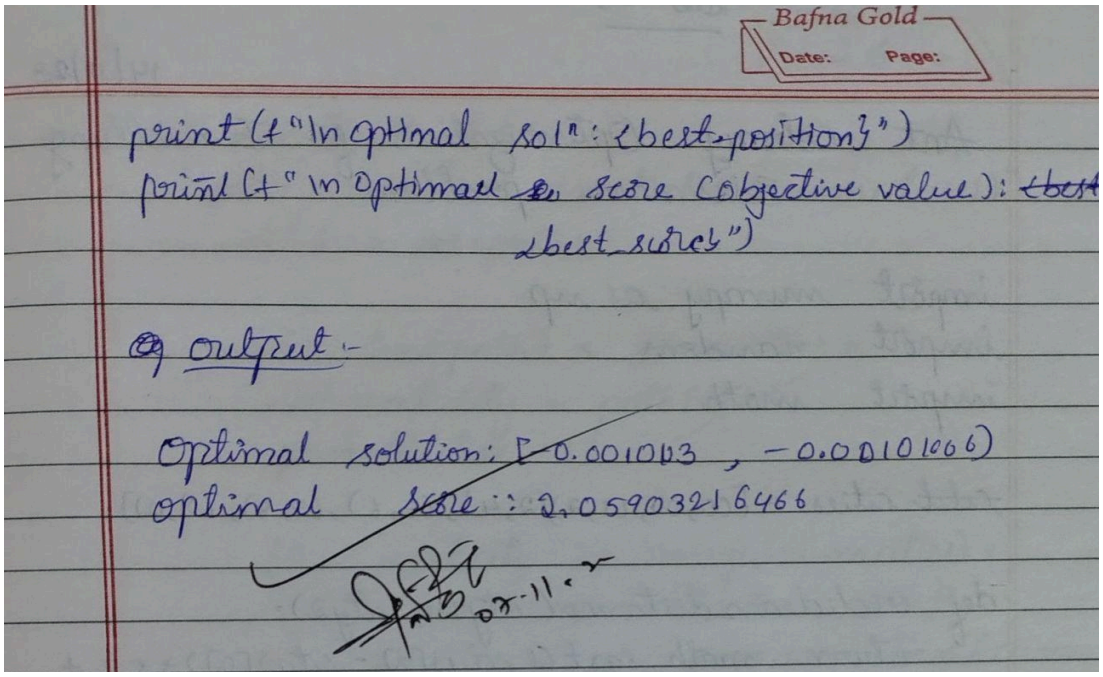
Page: 5

Particle Swarm Optimization

PSO is an Optimi

```
import numpy as np

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

class PSO:
    def __init__(self, func, dim, numparticles=30,
max_iter=100, w=0.5, c1=1.5, c2=1.5, bound=
        (-5.12, 5.12)):
    self.func = fun
    self.dim = dim
    self.numparticles = numparticles
    self.max_iter = max_iter
    self.w = w
    self.c1 = c1
    self.c2 = c2
    self.bound = bounds
    self.position = np.random.uniform(self.bounds[0],
        self.bounds[1], (self.numparticles, self.dim))
    self.velocities = np.random.uniform(-1,1,(self.
        numparticles, self.dim))
    self.personal_best_positions = np.copy(self.position)
    self.personal_best_scores = np.array([func(p)
        for p in self.positions])
    self.global_best_position = self.personal_best_position
        [np.argmin(self.personal_best_scores)]
    self.global_best_score = np.min(self.
        personalbest_score)
```

```
def update_velocity(self, i):
    r1, r2 = np.random.rand(2)
    cognitive_velocity = self.c1 * r1 * (self.personal_
        best_position[i] - self.positions[i])
    social_velocity = self.c2 * r2 * (self.global_best_
        position - self.positions[i])
    inertia_velocity = self.w * self.velocity[i]
    return interia_velocity + cognitive_velocity
        + social_velocity

def update_position(self, i)
    self.position[i] += self.velocities[i]
    self.position[i] = np.clip(self.positions[i],
        self.bounds[0], self.bounds[i])

def ~~upd~~optimize(self):
    for iteration in range(self.max_iter):
        for i in range(self.num_particles):
            fitness = self.func(self.position[i])
            if fitness < self.personal_best[i]
                self.personal_best[i] = fitness
                self.personal_best_position[i] = self.pos[i]

    for i in range(self.num_particles)
            self.velocities[i] = self.update_velocity(i)
            self.update_position(i)

    print("Iteration {iteration + 1}/{self.max_iter}
        BestScore: {self.global_best_score}")

best_position, best_score = pso.optimize()
```



Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Sphere function (fitness function)
def sphere_function(position):
    return np.sum(position**2)

# Parameters
num_particles = 30  # Number of particles
num_iterations = 100  # Number of iterations
dim = 2  # Dimensionality of the problem
bounds = [-10, 10]  # Search space bounds
inertia_weight = 0.7  # w
cognitive_coefficient = 1.5  # c1
social_coefficient = 1.5  # c2
tolerance = 1e-6  # Stopping tolerance for fitness

# Initialize particles
class Particle:
    def __init__(self):
        self.position = np.random.uniform(bounds[0], bounds[1], dim)  # Random position
        self.velocity = np.random.uniform(-1, 1, dim)  # Random velocity
        self.best_position = np.copy(self.position)  # Personal best position
        self.best_fitness = sphere_function(self.position)  # Personal best fitness
        self.fitness = self.best_fitness  # Current fitness
```

```
    def update_velocity(self, global_best_position):
        r1, r2 = np.random.rand(dim), np.random.rand(dim)
        cognitive_term = cognitive_coefficient * r1 * (self.best_position - self.position)
        social_term = social_coefficient * r2 * (global_best_position - self.position)
        self.velocity = inertia_weight * self.velocity + cognitive_term + social_term

    def update_position(self):
        self.position += self.velocity
        self.position = np.clip(self.position, bounds[0], bounds[1])  # Keep within bounds
        self.fitness = sphere_function(self.position)
        if self.fitness < self.best_fitness:  # Update personal best
            self.best_fitness = self.fitness
            self.best_position = np.copy(self.position)

# PSO implementation
def particle_swarm_optimization():
    particles = [Particle() for _ in range(num_particles)]
    global_best_position = particles[0].best_position
    global_best_fitness = particles[0].best_fitness
    fitness_history = []

    # Update global best from the initial population
    for particle in particles:
        if particle.best_fitness < global_best_fitness:
            global_best_fitness = particle.best_fitness
            global_best_position = np.copy(particle.best_position)

    for iteration in range(num_iterations):
        for particle in particles:
            particle.update_velocity(global_best_position)
            particle.update_position()

            # Update global best
            if particle.best_fitness < global_best_fitness:
                global_best_fitness = particle.best_fitness
                global_best_position = np.copy(particle.best_position)

        # Track global best fitness
        fitness_history.append(global_best_fitness)

        # Early stopping if fitness reaches tolerance
        if global_best_fitness <= tolerance:
            print(f"Converged at iteration {iteration}")
            break

    return global_best_position, global_best_fitness, fitness_history

# Run the PSO algorithm
```

```
best_position, best_fitness, fitness_history = particle_swarm_optimization()

# Print the results
print("Best position found:", best_position)
print("Best fitness achieved:", best_fitness)
```

Output:

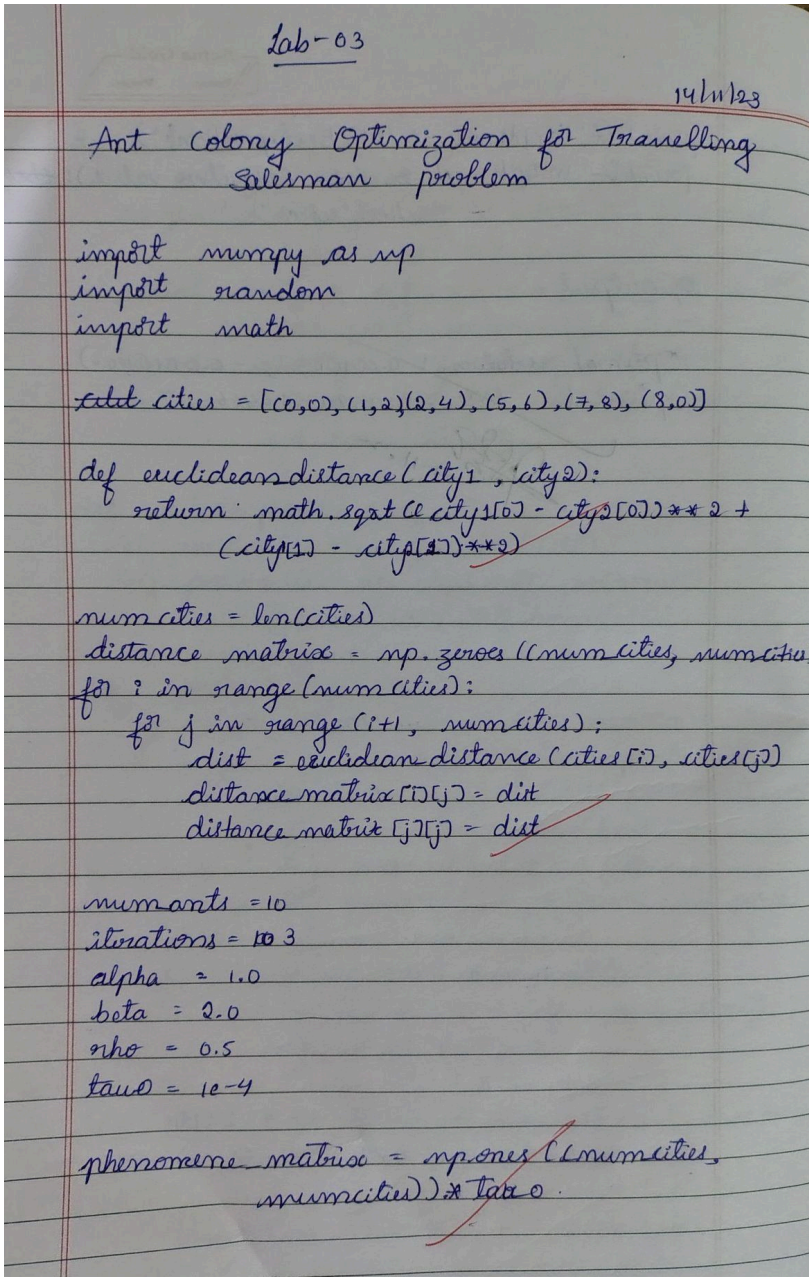
```
Converged at iteration 50
Best position found: [-0.00028244 -0.00092429]
Best fitness achieved: 9.340840427298652e-07
```

Program 3

Ant Colony Optimisation

Ants in nature deposit pheromones on their paths as they move. The intensity of the pheromone on a path influences the probability that other ants will choose that path. Over time, the pheromone trails strengthen on paths that are frequently used and weak on less frequently used ones. This behavior leads to the discovery of the shortest or optimal path between the ant colony and a food source. ACO mimics this process to solve various optimization problems, like the traveling salesman problem (TSP), vehicle routing problems, and others.

Algorithm:



Lab-03

14/11/23

Ant Colony Optimization for Travelling Salesman problem

```
import numpy as np
import random
import math

cities = [(0,0), (1,2), (2,4), (5,6), (7,8), (8,0)]

def euclidean_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 +
                     (city1[1] - city2[1])**2)

num_cities = len(cities)
distance_matrix = np.zeros((num_cities, num_cities
for i in range(num_cities):
    for j in range(i+1, num_cities):
        dist = euclidean_distance(cities[i], cities[j])
        distance_matrix[i][j] = dist
        distance_matrix[j][j] = dist

num_ants = 10
iterations = 100 3
alpha = 1.0
beta = 2.0
rho = 0.5
tau0 = 1e-4

pheromone_matrix = np.ones((num_cities,
                   num_cities)) * tau0
```

Bafna Gold
Date: 6 Page:

```
def construct_sol():
    path = [random.randint(0, numcities-1)]
    visited = set(path)

    while len(path) < numcities:
        current_city = path[-1]
        probabilities = []

        for next_city in range(numcities):
            if next_city not in visited:
                pheromone = pheromone_matrix[current
_city][next_city] ** alpha
                heuristic = (1.0 / distance_matrix[current
_city][next_city]) ** beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)

        total_prob = sum(probabilities)
        probabilities = [p / total_prob for p in probabiliti
                                            ties]
        next_city = np.random.choice(range(numcities),
                p = probabilities)
        path.append(next_city)
        visited.add(next_city)

    return path

def update_pheromones(all_paths, all_lengths):
    global pheromone_matrix
    pheromone_matrix *= (1-rho)
```

```
for path, length in zip(all_paths, all_lengths)
    pheromone_deposite = 1.0 / length

best_path = None
best_loop = float('inf')

for iteration in range(iterations)
    all_paths = []
    all_lengths = []

for _ in range(num_ants)
    path = construct_sol()

    all_paths.append(path)
    all_length.append(length)

    if length < best_length
        best_length = length
        best_path = path

update_pheromones(all_paths, all_length)

print(f"Iteration {iteration+1}/{iterations}:
        best_length={best_length}")

print("\nBest path found:", best_path)
print("Best length:", best_length)
```

O/P

```
Iteration 1/3 : Best length = 26.96837...
Iteration 2/3 : Best length = 26.96837...
Iteration 3/3 : Best length = 26.96837...
Best path found : [0, 1, 2, 3, 4, 5]
Best length : 26.96837...
```

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import euclidean

# Define the problem: coordinates of cities
cities = np.array([
    [0, 0], [2, 3], [5, 2], [6, 6], [8, 3],
    [1, 5], [4, 7], [7, 8], [9, 5], [3, 1]
])
num_cities = len(cities)

# Parameters
num_ants = 10
num_iterations = 100
alpha = 1  # Importance of pheromone
beta = 2   # Importance of heuristic (1/distance)
rho = 0.5  # Pheromone evaporation rate
initial_pheromone = 1.0

# Distance matrix
distances = np.array([[euclidean(cities[i], cities[j]) for j in range(num_cities)] for i in
range(num_cities)])

# Heuristic information (1 / distance), avoiding division by zero
heuristics = np.zeros_like(distances)  # Initialize as zero
for i in range(num_cities):
    for j in range(num_cities):
        if i != j:
            heuristics[i, j] = 1 / distances[i, j]  # Only calculate for non-diagonal elements
        else:
            heuristics[i, j] = 0  # Set diagonal to zero (no heuristic for the same city)

# Pheromone matrix
pheromones = np.full((num_cities, num_cities), initial_pheromone)

# Ant class
class Ant:
    def __init__(self):
        self.visited = []
        self.total_distance = 0

    def choose_next_city(self, current_city):
        probabilities = []
        for city in range(num_cities):
            if city not in self.visited:
                pheromone = pheromones[current_city][city] ** alpha
```

```
                heuristic = heuristics[current_city][city] ** beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)

        # Convert to numpy array for easier manipulation
        probabilities = np.array(probabilities)

        # Check if all probabilities are zero, avoid NaN
        if probabilities.sum() == 0:
            unvisited_cities = [city for city in range(num_cities) if city not in self.visited]
            return np.random.choice(unvisited_cities)

        # Normalize probabilities to make sure they sum to 1
        probabilities /= probabilities.sum()
        return np.random.choice(range(num_cities), p=probabilities)

    def complete_tour(self):
        # Complete the tour by returning to the start city
        self.total_distance += distances[self.visited[-1]][self.visited[0]]
        self.visited.append(self.visited[0])

# ACO implementation
def ant_colony_optimization():
    global pheromones
    best_solution = None
    best_distance = float('inf')
    best_history = []

    for iteration in range(num_iterations):
        all_ants = []

        # Each ant constructs a solution
        for _ in range(num_ants):
            ant = Ant()
            current_city = np.random.randint(0, num_cities)  # Start at a random city
            ant.visited.append(current_city)

            while len(ant.visited) < num_cities:
                next_city = ant.choose_next_city(current_city)
                ant.total_distance += distances[current_city][next_city]
                ant.visited.append(next_city)
                current_city = next_city

            # Complete the tour
            ant.complete_tour()
            all_ants.append(ant)

        # Update global best
```

```
        for ant in all_ants:
            if ant.total_distance < best_distance:
                best_distance = ant.total_distance
                best_solution = ant.visited

        # Update pheromone trails
        pheromones *= (1 - rho)  # Evaporation
        for ant in all_ants:
            for i in range(num_cities):
                from_city = ant.visited[i]
                to_city = ant.visited[i + 1] if i + 1 < len(ant.visited) else ant.visited[0]
                pheromones[from_city][to_city] += 1 / ant.total_distance

        # Track the best distance in history
        best_history.append(best_distance)
        print(f"Iteration {iteration + 1}, Best Distance: {best_distance}")

    return best_solution, best_distance, best_history

# Run the ACO algorithm
best_solution, best_distance, best_history = ant_colony_optimization()

# Print the results
print("\nBest route found:", best_solution)
print("Shortest distance:", best_distance)
```

Output:

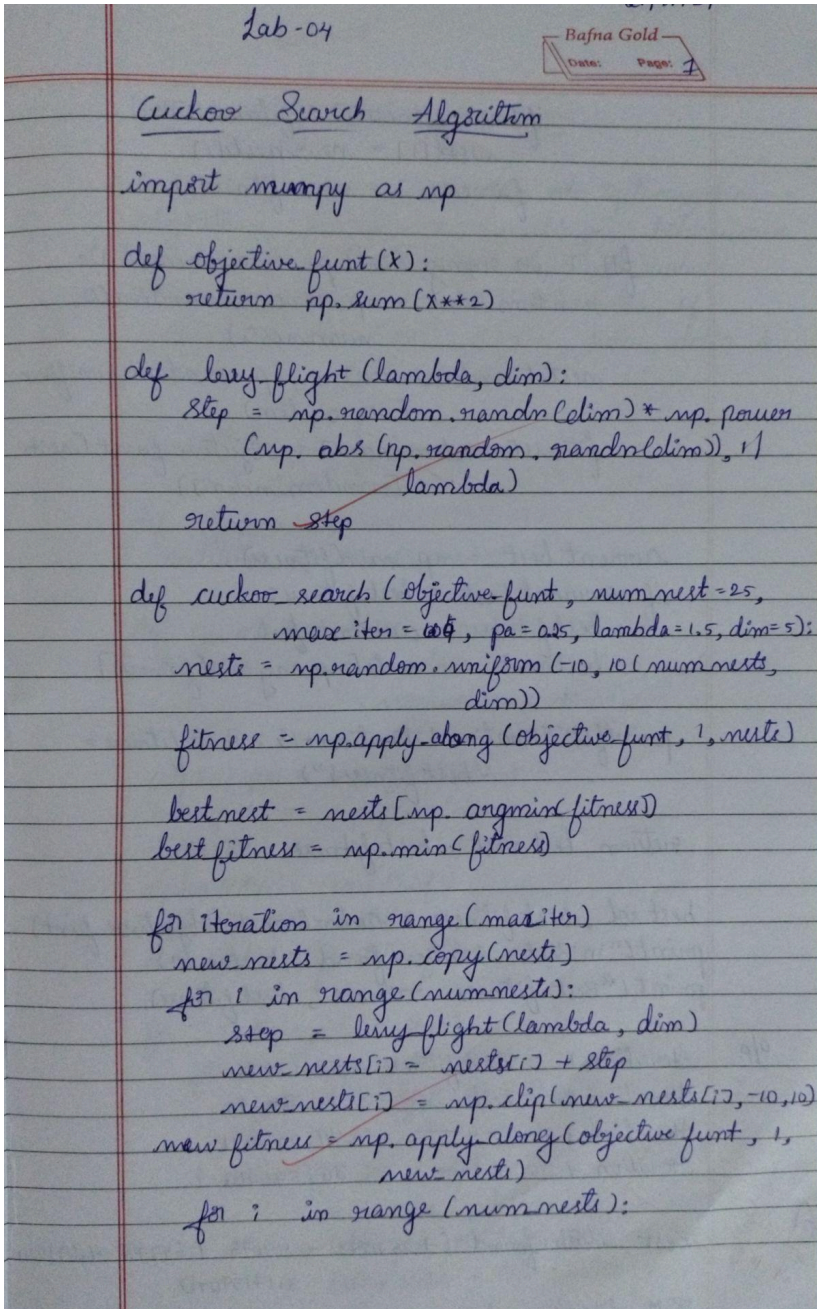
```
Best route found: [1, 5, 6, 3, 7, 8, 4, 2, 9, 0, 1]
Shortest distance: 28.321549034227672
```

Program 4

Cuckoo Search(CS)

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behaviour involves laying eggs in the nests 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. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:



Lab-04

Bafna Gold

Date: Page: 1

Cuckoo Search Algorithm

```
import numpy as np

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

def levy_flight(lambda, dim):
    step = np.random.randn(dim) * np.power
        (np.abs(np.random.randn(dim)), 1/
                lambda)
    return step

def cuckoo_search(objective_funt, num_nest=25,
        max_iter=100, pa=0.25, lambda=1.5, dim=5):
    nests = np.random.uniform(-10, 10(num_nests,
                dim))
    fitness = np.apply_along(objective_funt, 1, nests)

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

    for iteration in range(max_iter)
        new_nests = np.copy(nests)
        for i in range(num_nests):
            step = levy_flight(lambda, dim)
            new_nests[i] = nests[i] + step
            new_nests[i] = np.clip(new_nests[i], -10, 10)
    new_fitness = np.apply_along(objective_funt, 1,
                new_nests)
    for i in range(num_nests):
```

```
if (new_fitness[i] < fitness[i]):
    nests[i] = new_nests[i]
    fitness[i] = new_fitness[i]

for i in range (int(pa * num_nests)):
    random_index = np.random.randint(0,
                   num_nests)
    nests[random_index] = np.random.uniform
                   (-10,10, dim)
    fitness[random_index] = objective_funct(nests
                   [random_index])

current_best = np.min(fitness)
if current_best < best_fitness:
    best_fitness = current_best
    best_nest = nests[np.argmin(fitness)]

printf(f"Iteration {iteration+1}: Best fitness =
                   {best_fitness}")

return best_nest, best_fitness

best_sol, best_fitness = cuckoo_search(objective_funct)
print("\n Best solution found:", best_sol)
print("Best fitness value:", best_fitness)
```

O/P

```
Iteration 1 : Best fitness : 58.03826
Iteration 2 : Best  -"-  : 44.467699
Iteration 3 :  -"-  : 44.497991
Iteration 4 :  -"-  : 36.590463

Best solution found : [-1.26757 -0.44173 1.64976 4.69124
                       3.17159078]
Best fitness
```

Code:

```
import numpy as np
import math  # Import the standard math module

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

def cuckoo_search(obj_function, bounds, n=25, pa=0.25, max_iter=100):
    # Initialize nests
    dim = len(bounds)
    nests = np.random.rand(n, dim)
    for i in range(dim):
        nests[:, i] = nests[:, i] * (bounds[i][1] - bounds[i][0]) + bounds[i][0]
    fitness = np.array([obj_function(nest) for nest in nests])

    # Start optimization
    for _ in range(max_iter):
        for i in range(n):
            # Generate a new solution via Levy flight
            new_nest = nests[i] + levy_flight(1.5) * np.random.randn(dim)
            # Apply bounds
            new_nest = np.clip(new_nest, [b[0] for b in bounds], [b[1] for b in bounds])
            new_fitness = obj_function(new_nest)
            # Update if new solution is better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

        # Abandon some nests and create new ones
        abandon_idx = np.random.rand(n) < pa
        for i in np.where(abandon_idx)[0]:
            nests[i] = np.random.rand(dim) * (np.array([b[1] for b in bounds]) - np.array([b[0] for b in
bounds])) + np.array([b[0] for b in bounds])
            fitness[i] = obj_function(nests[i])

    # Return the best solution
    best_idx = np.argmin(fitness)
    return nests[best_idx], fitness[best_idx]

# Example usage: Minimize f(x) = x^2
```

```
def objective(x):
    return sum(xi**2 for xi in x)

bounds = [(-10, 10), (-10, 10)]  # 2D problem
best_solution, best_fitness = cuckoo_search(objective, bounds)
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

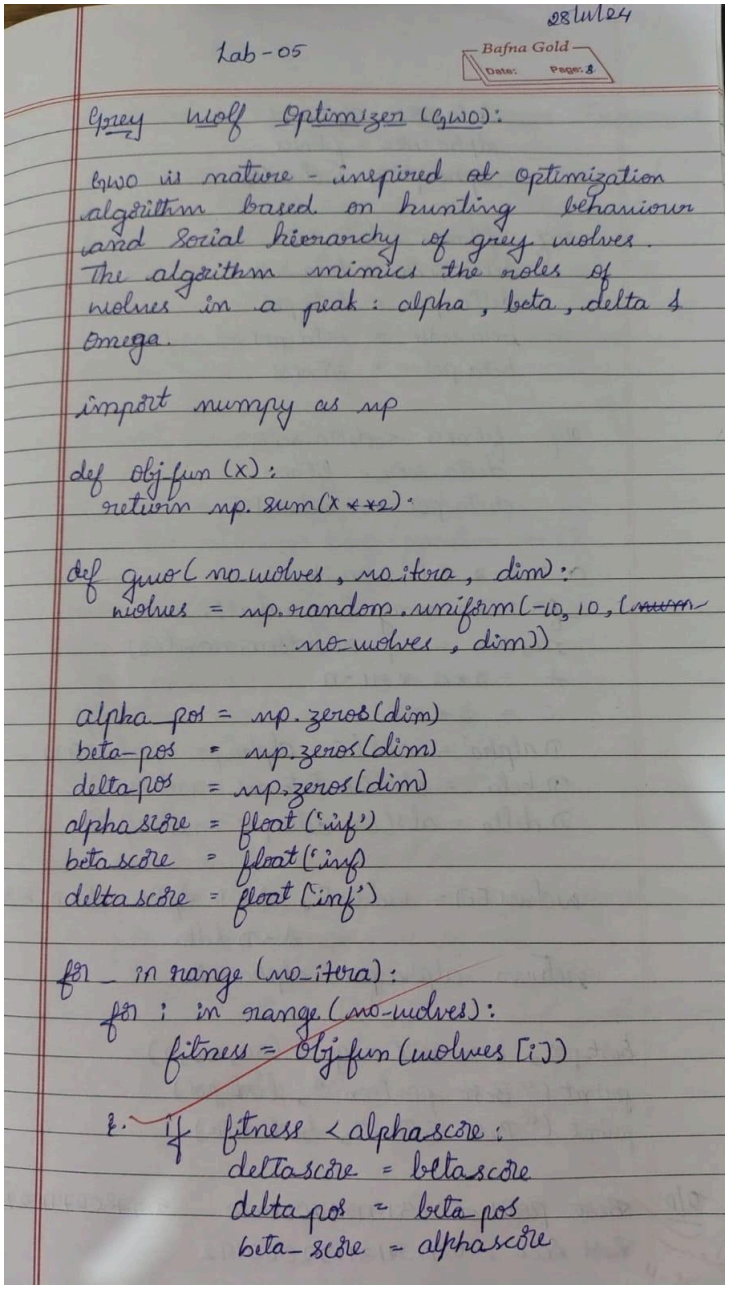
```
Best Solution: [0.20618515 0.76221114]
Best Fitness: 0.6234781374664989
```

Program 5

Grey Wolf Optimiser:

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behaviour of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:



28/11/24

Lab - 05

Bafna Gold

Date: Page: 8

Grey Wolf Optimizer (GWO):

GWO is nature - inspired optimization algorithm based on hunting behaviour and social hierarchy of grey wolves. The algorithm mimics the roles of wolves in a peak : alpha, beta, delta & omega.

```
import numpy as np

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

def gwo(no_wolves, no_itera, dim):
    wolves = np.random.uniform(-10, 10, (
                no_wolves, dim))

    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')

    for _ in range(no_itera):
        for i in range(no_wolves):
            fitness = obj_fun(wolves[i])

            if fitness < alpha_score:
                delta_score = beta_score
                delta_pos = beta_pos
                beta_score = alpha_score
```

```
        beta_pos = alpha_pos
        alpha_score = fitness
        alpha_pos = wolves[i]

    elif fitness < beta_score:
        delta_score = beta_score
        delta_pos = beta_pos
        beta_score = beta_pos
        beta_pos = fitness

    elif fitness < delta_score:
        delta_score = fitness
        delta_pos = wolves[i]

a = 2 - 2 * (_ / no_itera)
for i in range (no_wolves):
r1, r2 = np.random.rand(2)
A = 2*a*r1 - a
C = 2*r2
D_alpha = abs(C * alpha_pos - wolves[i])
D_beta = abs(C* beta_pos - wolves[i])
D_delta = abs(C* delta_pos - wolves[i])

wolves[i] = wolves[i] - A*D_alpha - A*D_beta -
                        A*D_delta
return alpha_pos, alpha_score

best_pos, best_score = gwo(30, 100, 2)
print("Best position:", best_pos)
print("Best score:", best_score)
```

O/P: Best position: [3.19189700[illegible]30 5.77800776e-29]
Best Score: 0.9939113113200019 2

Code:

```
import numpy as np

def objective_function(x):
    """Example objective function: Sphere function."""
    return sum(x**2)

def initialize_population(dim, n_wolves, bounds):
    """Initialize the positions of the wolves randomly
    within the given bounds."""
    return np.random.uniform(bounds[0], bounds[1], (n_wolves, dim))

def gwo(objective_function, bounds, dim, n_wolves, n_iterations):
    """
    Grey Wolf Optimizer (GWO) algorithm.
    Parameters:
    - objective_function: The function to optimize.
    - bounds: Tuple of (lower_bound, upper_bound) for each dimension.
    - dim: Number of dimensions.
    - n_wolves: Number of wolves in the pack.
    - n_iterations: Number of iterations.
    Returns:
    - best_solution: The best solution found.
    - best_score: The best objective function value.
    """
    # Initialize population
    wolves = initialize_population(dim, n_wolves, bounds)
    fitness = np.apply_along_axis(objective_function, 1, wolves)

    # Initialize alpha, beta, and delta
    alpha, beta, delta = np.argsort(fitness)[:3]
    alpha_pos, alpha_score = wolves[alpha], fitness[alpha]
    beta_pos, beta_score = wolves[beta], fitness[beta]
    delta_pos, delta_score = wolves[delta], fitness[delta]

    # Main optimization loop
    for iteration in range(n_iterations):
        a = 2 - 2 * (iteration / n_iterations)  # Linearly decreasing a
        for i in range(n_wolves):
            for j in range(dim):
                # Update each wolf's position
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolves[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] - wolves[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] - wolves[i, j])
            X3 = delta_pos[j] - A3 * D_delta

            # Average position update
            wolves[i, j] = (X1 + X2 + X3) / 3.0

        # Enforce bounds
        wolves[i, :] = np.clip(wolves[i, :], bounds[0], bounds[1])

    # Evaluate fitness and update alpha, beta, delta
    fitness = np.apply_along_axis(objective_function, 1, wolves)
    sorted_indices = np.argsort(fitness)
    alpha, beta, delta = sorted_indices[:3]
    alpha_pos, alpha_score = wolves[alpha], fitness[alpha]
    beta_pos, beta_score = wolves[beta], fitness[beta]
    delta_pos, delta_score = wolves[delta], fitness[delta]

  return alpha_pos, alpha_score

# Example usage
dim = 5  # Number of dimensions
bounds = (-10, 10)  # Search space bounds
n_wolves = 30  # Number of wolves
n_iterations = 100  # Number of iterations

best_solution, best_score = gwo(objective_function, bounds, dim, n_wolves, n_iterations)
print(f"Best solution: {best_solution}")
print(f"Best score: {best_score}")
```

Output:

```
Best solution: [-1.52807921e-11 -1.39104785e-11  1.29132014e-11 -1.85709387e-11
 -1.49726055e-11]
Best score: 1.16281346713889e-21
```

Program 6

Parallel Cellular Algorithms and Programs

The Parallel Cell Algorithm is a computational method used for solving problems that involve large datasets, spatial partitioning, or simulations where a domain is divided into smaller "cells" that can be processed independently or semi-independently in parallel. It is commonly applied in scientific computing, numerical simulations, and artificial intelligence, where computational efficiency is crucial.

Algorithm:

Lab-06

Rafna Gold

Date: Page: 9

Parallel cellular Algorithms and Programs

Parallel cellular algorithm are inspired by functioning of biological cells that operate in highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently.

```
import numpy as np

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

num_cells =100
grid_size = (10,10)
solution_dim = 2
iteration =100
neighbor_radius = 1
search_space_bounds = [-5.0, 5.0]

def evaluate_fitness(population):
    fitness = np.zeros((grid_size[0], grid_size[0]))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            fitness [i, j] = fitness_function(popu
                -lation[i, j])
    return fatse fitness
```

```
def get_neighbors(grid_size, i, j, radius):
    neighbours = []
    for di in range(-radius, radius+1):
        for dj in range(-radius, radius+1):
            ni, nj = (i+di) % grid_size[0], (j+dj) %
                     grid_size[1]
            if (di != 0 or dj != 0):
                neighbors.append((ni, nj))
    return neighbors

def update_population(population, fitness, grid_size,
                      radius):
    new_population = np.copy(population)
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            neighbors = get_neighbors(grid_size, i, j,
                                      radius)
            best_neighbor = population[i, j]
            best_fitness = fitness[i, j]
            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_neighbor = population[ni, nj]
                    best_fitness = fitness[ni, nj]
            new_population[i, j] = (population[i, j] +
                                    best_neighbor) / 2.0
    return new_population

def parallel_cellular():
    population = initialize_population(grid_size,
                 solution_dim, search_space_bounds)
    best_solution = None
    best_fitness = float('inf')
```

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

  population = update_population(population,
      fitness, grid_size, neighborhood_radius)

  print(f"Iteration {iteration+1}/{iterations},
   Best fitness: {best_fitness:.6f}")

print("\nBest Solution found :", best_solution)
print("Best fitness value :", best_fitness)
```

Output :-

```
Best Solution found : [4.52424603e-06 -3.90935704e-0
Best fitness value : 3.5751874628488416-11
```

Code:

```
import numpy as np

# Define the mathematical function to optimize (e.g., Sphere function)
def objective_function(x, y):
    return x**2 + y**2  # Minimize this function

# Define the problem parameters
grid_size = 10  # Size of the 2D grid
iterations = 100  # Number of iterations
neighborhood_size = 1  # Neighborhood radius

# Initialize the population of cells (random positions in the solution space)
def initialize_population(grid_size, lower_bound, upper_bound):
    grid = np.random.uniform(lower_bound, upper_bound, (grid_size, grid_size, 2))  # (x, y) for each
cell
    return grid

# Evaluate the fitness of each cell
def evaluate_fitness(grid):
    fitness = np.zeros((grid_size, grid_size))
    for i in range(grid_size):
        for j in range(grid_size):
            x, y = grid[i, j]
            fitness[i, j] = objective_function(x, y)
    return fitness

# Update the state of each cell based on its neighborhood
def update_states(grid, fitness):
    new_grid = grid.copy()
    for i in range(grid_size):
        for j in range(grid_size):
            # Get the neighborhood
            neighborhood = []
            for di in range(-neighborhood_size, neighborhood_size + 1):
                for dj in range(-neighborhood_size, neighborhood_size + 1):
                    ni, nj = (i + di) % grid_size, (j + dj) % grid_size
                    neighborhood.append((fitness[ni, nj], grid[ni, nj]))

            # Find the best neighbor
            best_neighbor = min(neighborhood, key=lambda x: x[0])
            new_grid[i, j] = best_neighbor[1]  # Update to the best neighbor's position

    return new_grid

# Main function to execute the Parallel Cellular Algorithm
```

```
def parallel_cellular_algorithm():
    # Initialize parameters
    lower_bound, upper_bound = -10, 10  # Solution space bounds
    grid = initialize_population(grid_size, lower_bound, upper_bound)
    best_solution = None
    best_fitness = float('inf')

    for _ in range(iterations):
        fitness = evaluate_fitness(grid)

        # Update best solution
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_index = np.unravel_index(fitness.argmin(), fitness.shape)
            best_solution = grid[best_index]

        # Update the states of the grid
        grid = update_states(grid, fitness)

    return best_solution, best_fitness

# Run the algorithm
best_solution, best_fitness = parallel_cellular_algorithm()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

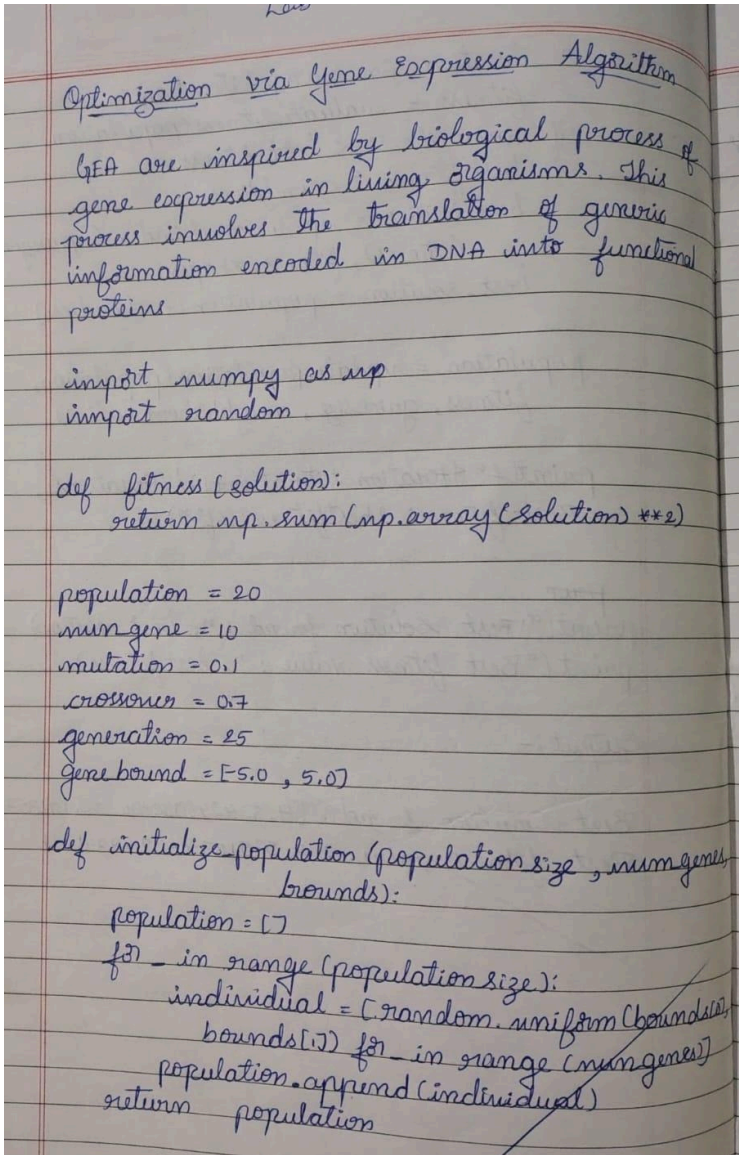
```
Best solution: [-1.52807921e-11 -1.39104785e-11  1.29132014e-11 -1.85709387e-11
 -1.49726055e-11]
Best score: 1.16281346713889e-21
```

Program 7

Optimization via Gene Expression Algorithms

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:

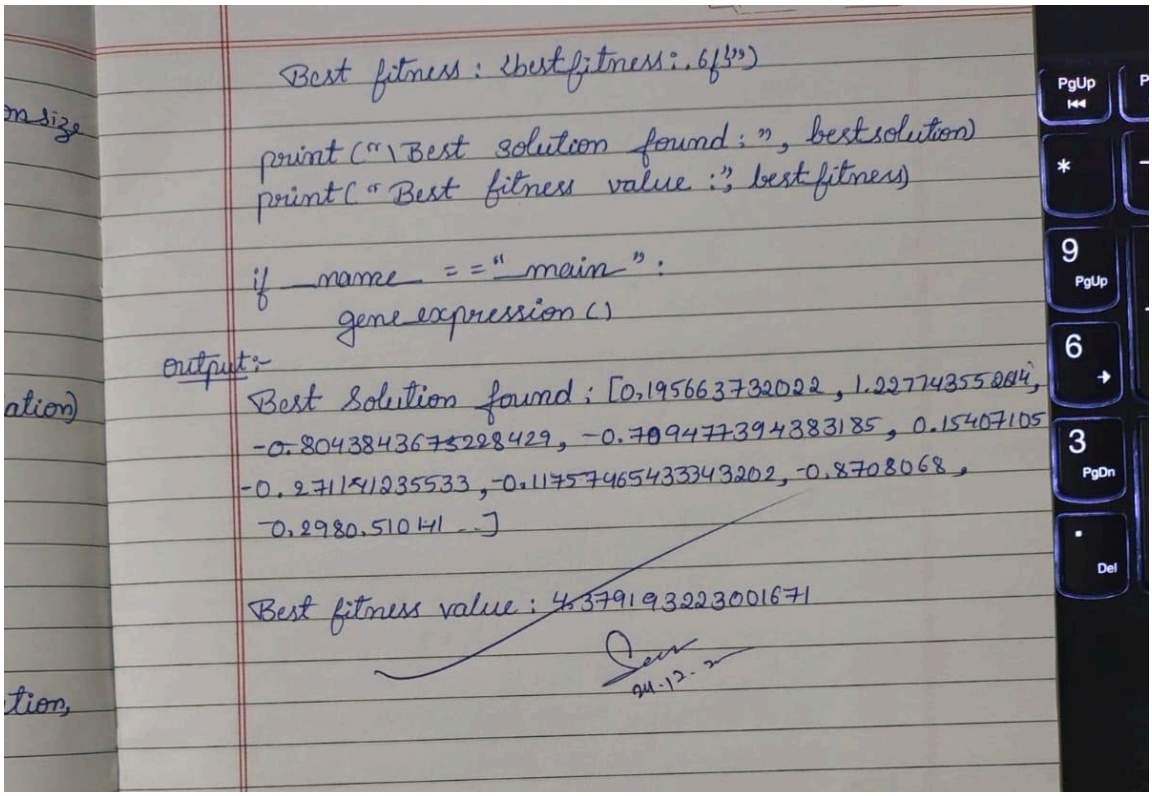


Optimization via Gene Expression Algorithm

GEA are inspired by biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins

```
import numpy as np
import random

def fitness(solution):
    return np.sum(np.array(solution)**2)

population = 20
num_gene = 10
mutation = 0.1
crossover = 0.7
generation = 25
gene_bound = [-5.0, 5.0]

def initialize_population(population_size, num_genes,
                          bounds):
    population = []
    for _ in range(population_size):
        individual = [random.uniform(bounds[0],
            bounds[1]) for _ in range(num_genes)]
        population.append(individual)
    return population
```

```
def evaluate_population(population):
    fitness_scores = [fitness_for(individual)
        for individual in population]
    return fitness_scores

def tournament_se(population, fitness_scores,
                  tournament_size=3):
    selected = []
    for _ in range(len(population)):
        participants = random.sample(list(enum
            -erate(fitness_scores)), tournment_size)
        winner = min(participants, key = lambda
            x: x[1])
        selected.append(population[winner[0]])
    return selected

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        point = random.randint(1, len(parent1)-1)
        offspring1 = parent1[:point] + parent2[point:]
        offspring2 = parent2[:point] + parent1[point:]
        return offspring1, offspring2
    return parent1, parent2

def mutate(individual, bounds, mutationrate):
    for i in range(len(individual)):
        if random.random() < mutationrate
            individual[i] = random.uniform
                (bound[0], bound[1])
    return individual
```



Code:

```
import numpy as np

# Define the mathematical function to optimize (e.g., Sphere function)
def objective_function(x):
    return np.sum(x**2)  # Minimize this function

# Initialize parameters
population_size = 50  # Number of genetic sequences in the population
num_genes = 10  # Number of genes in each sequence
mutation_rate = 0.1  # Probability of mutation
crossover_rate = 0.8  # Probability of crossover
generations = 100  # Number of generations
lower_bound, upper_bound = -10, 10  # Solution space bounds

# Initialize the population with random genetic sequences
def initialize_population():
    return np.random.uniform(lower_bound, upper_bound, (population_size, num_genes))

# Evaluate the fitness of each genetic sequence
def evaluate_fitness(population):
    fitness = np.array([objective_function(individual) for individual in population])
    return fitness

# Selection: Select genetic sequences based on fitness (tournament selection)
```

```
def select_parents(population, fitness):
    selected_parents = []
    for _ in range(population_size):
        candidates = np.random.choice(len(population), 3, replace=False)
        best_candidate = candidates[np.argmin(fitness[candidates])]
        selected_parents.append(population[best_candidate])
    return np.array(selected_parents)

# Crossover: Perform crossover between selected sequences to produce offspring
def crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        parent1 = parents[i]
        parent2 = parents[(i + 1) % len(parents)]
        if np.random.rand() < crossover_rate:
            crossover_point = np.random.randint(1, num_genes - 1)
            child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
            child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
            offspring.extend([child1, child2])
        else:
            offspring.extend([parent1, parent2])
    return np.array(offspring)

# Mutation: Apply mutation to the offspring
def mutate(offspring):
    for individual in offspring:
        if np.random.rand() < mutation_rate:
            gene_to_mutate = np.random.randint(num_genes)
            individual[gene_to_mutate] = np.random.uniform(lower_bound, upper_bound)
    return offspring

# Gene Expression: Decode the genetic sequence into a functional solution (identity in this case)
def gene_expression(sequence):
    return sequence

# Main function to execute the Gene Expression Algorithm
def gene_expression_algorithm():
    population = initialize_population()
    best_solution = None
    best_fitness = float('inf')

    for generation in range(generations):
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.argmin(fitness)]
```

```
        # Selection, Crossover, Mutation, and Gene Expression
        parents = select_parents(population, fitness)
        offspring = crossover(parents)
        mutated_offspring = mutate(offspring)
        population = np.array([gene_expression(ind) for ind in mutated_offspring])

    return best_solution, best_fitness

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Best Solution: [ 6.02069591e-04  6.33325442e-02  5.84702655e-01 -2.23247895e-01
 -3.99173913e-01  2.32991980e-01 -6.35082961e-01  1.57878812e-01
  4.75012210e-02  2.92306981e-01]
Best Fitness: 1.1253090907692773
```