VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sneha N Shastri (1BM22CS283)

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 **Sneha N Shastri (1BM22CS283),** who is a 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 A N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

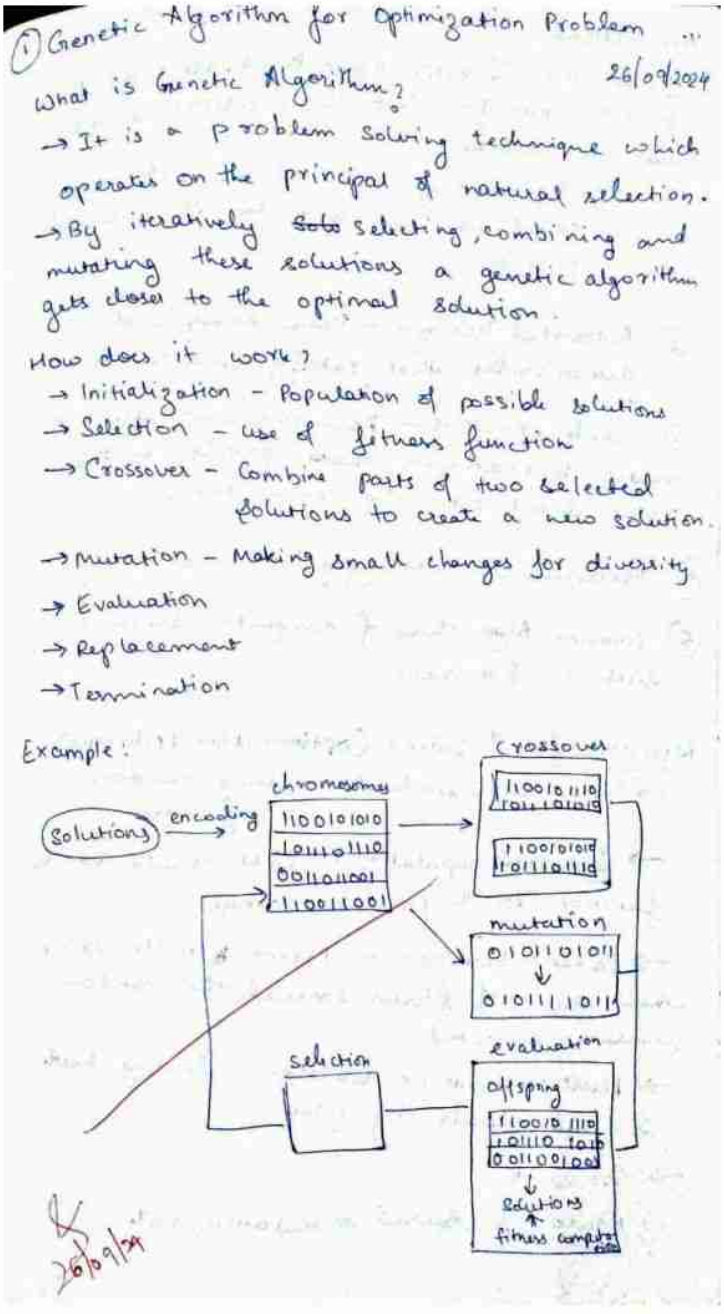
Index

Github Link:
https://github.com/snehanshastri/BIS

Program 1

Genetic Algorithm for Optimization Problems: Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:



Code:

```
import numpy as np
import random

# Define the function to maximize
def objective_function(x):
    return x ** 2

# Generate an initial population
def initialize_population(pop_size, bounds):
    return [random.uniform(bounds[0], bounds[1]) for _ in range(pop_size)]

# Selection: Tournament selection - best individuals are selected as parent
def select_parents(population, scores):
    tournament_size = 3
    selected = random.sample(list(zip(population, scores)), tournament_size)
    selected.sort(key=lambda x: x[1], reverse=True)
    return selected[0][0], selected[1][0]

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

# Mutation: Randomly perturb the child
def mutate(child, mutation_rate, bounds):
    if random.random() < mutation_rate:
        child += random.uniform(-1, 1)  # Random perturbation - adding a random value
        child = np.clip(child, bounds[0], bounds[1])  # Keep within bounds
    return child

# Genetic Algorithm
def genetic_algorithm(pop_size, bounds, mutation_rate, generations):
    # Initialize population
    population = initialize_population(pop_size, bounds)

    for generation in range(generations):
        scores = [objective_function(x) for x in population]
        next_generation = []

        # Create the next generation
        for _ in range(pop_size // 2):  # Create pairs of parents
```

```
            parent1, parent2 = select_parents(population, scores)
            child1 = crossover(parent1, parent2)
            child2 = crossover(parent1, parent2)

            # Mutate children
            child1 = mutate(child1, mutation_rate, bounds)
            child2 = mutate(child2, mutation_rate, bounds)

            next_generation.extend([child1, child2])

        population = next_generation

    # Get the best solution
    scores = [objective_function(x) for x in population]
    best_index = np.argmax(scores)
    best_solution = population[best_index]
    best_score = scores[best_index]

    return best_solution, best_score

# Parameters
population_size = 100
bounds = (-10, 10)  # Updated bounds
mutation_rate = 0.1 #indicates the number of mutations - exploration of space
generations = 200 #determines the number of iterations

# Run the Genetic Algorithm
best_solution, best_score = genetic_algorithm(population_size, bounds, mutation_rate,
generations)
print(f"Best solution: x = {best_solution}, f(x) = {best_score}")
```

Output:

```
Best solution: x = 10.0, f(x) = 100.0
```

Program 2

Particle Swarm Optimization for Function Optimization: Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

07/11/2024 Particle Swarm Optimization

Steps involved:

1. Define objective function
2. Create a particle class – self.position, self.velocity – current position and velocity randomly initialized
   self.best_position, self.best_value best position and value of objective function reached so far
3. Evaluate for each particle
   Update personal best and global best
4. Update velocity and position based on w, c1, c2

Code:

```
import numpy as np

def objective_function(x):
    return x**2

#Define Particle class
class Particle:
    def __init__(self, dim, bounds):
        #Initialize particle's position and velocity within bounds
        self.position = np.random.uniform(bounds[0],
                                          bounds[1], dim)
        self.velocity = np.random.uniform(-1, 1, dim)
        self.best_position = np.copy(self.position)
```

Code:

```
import numpy as np

# Define the objective function to be maximized (we want to maximize x^2)
def objective_function(x):
    return x**2

# Define the Particle class
class Particle:
    def __init__(self, dim, bounds):
        # Initialize particle's position and velocity within the bounds
        self.position = np.random.uniform(bounds[0], bounds[1], dim)
        self.velocity = np.random.uniform(-1, 1, dim)  # Initialize velocity
        self.best_position = np.copy(self.position)  # Best position so far
        self.best_value = objective_function(self.position)  # Best value so far

# Particle Swarm Optimization Algorithm
def particle_swarm_optimization(objective_function, bounds, num_particles=30,
max_iter=100):
    # Initialize parameters
    dim = 1  # We are optimizing over 1 dimension (for this example, x^2)
    w = 0.5  # Inertia weight (balances exploration and exploitation)
    c1 = 1.5  # Cognitive (personal) weight
    c2 = 1.5  # Social (global) weight

    # Initialize particles
    particles = [Particle(dim, bounds) for _ in range(num_particles)]
    global_best_position = None
    global_best_value = -np.inf  # We want to maximize, so start with a very low value

    # Main PSO loop
    for iteration in range(max_iter):
        for particle in particles:
            # Evaluate the particle's fitness (value of the objective function at current position)
            current_value = objective_function(particle.position)

            # Update the personal best if the current position is better
            if current_value > particle.best_value:
                particle.best_position = np.copy(particle.position)
                particle.best_value = current_value

            # Update the global best if the current position is better
```

```
            if current_value > global_best_value:
                global_best_position = np.copy(particle.position)
                global_best_value = current_value

        # Update particle velocities and positions
        for particle in particles:
            r1, r2 = np.random.rand(2)  # Random coefficients for exploring and exploiting
            particle.velocity = (w * particle.velocity +
                                 c1 * r1 * (particle.best_position - particle.position) +
                                 c2 * r2 * (global_best_position - particle.position))
            particle.position = particle.position + particle.velocity  # Update position

            # Ensure the position stays within bounds (-10 to 10)
            particle.position = np.clip(particle.position, bounds[0], bounds[1])

        # Print the current global best value for monitoring progress
        print(f'Iteration {iteration + 1}: Global Best Value = {global_best_value}")

    return global_best_position, global_best_value

# Set bounds for the optimization (-10 to 10)
bounds = (-10, 10)

# Run PSO to find the maximum of f(x) = x^2
best_position, best_value = particle_swarm_optimization(objective_function, bounds)

# Output the final result
print(f"\nGlobal best position: {best_position}")
print(f'Global best value: {best_value}")
```

Output:

```
Iteration 1: Global Best Value = [85.58647018]
Iteration 2: Global Best Value = [100.]
Iteration 3: Global Best Value = [100.]
Iteration 4: Global Best Value = [100.]
Iteration 5: Global Best Value = [100.]
Iteration 6: Global Best Value = [100.]
Iteration 7: Global Best Value = [100.]
Iteration 8: Global Best Value = [100.]
Iteration 9: Global Best Value = [100.]
```

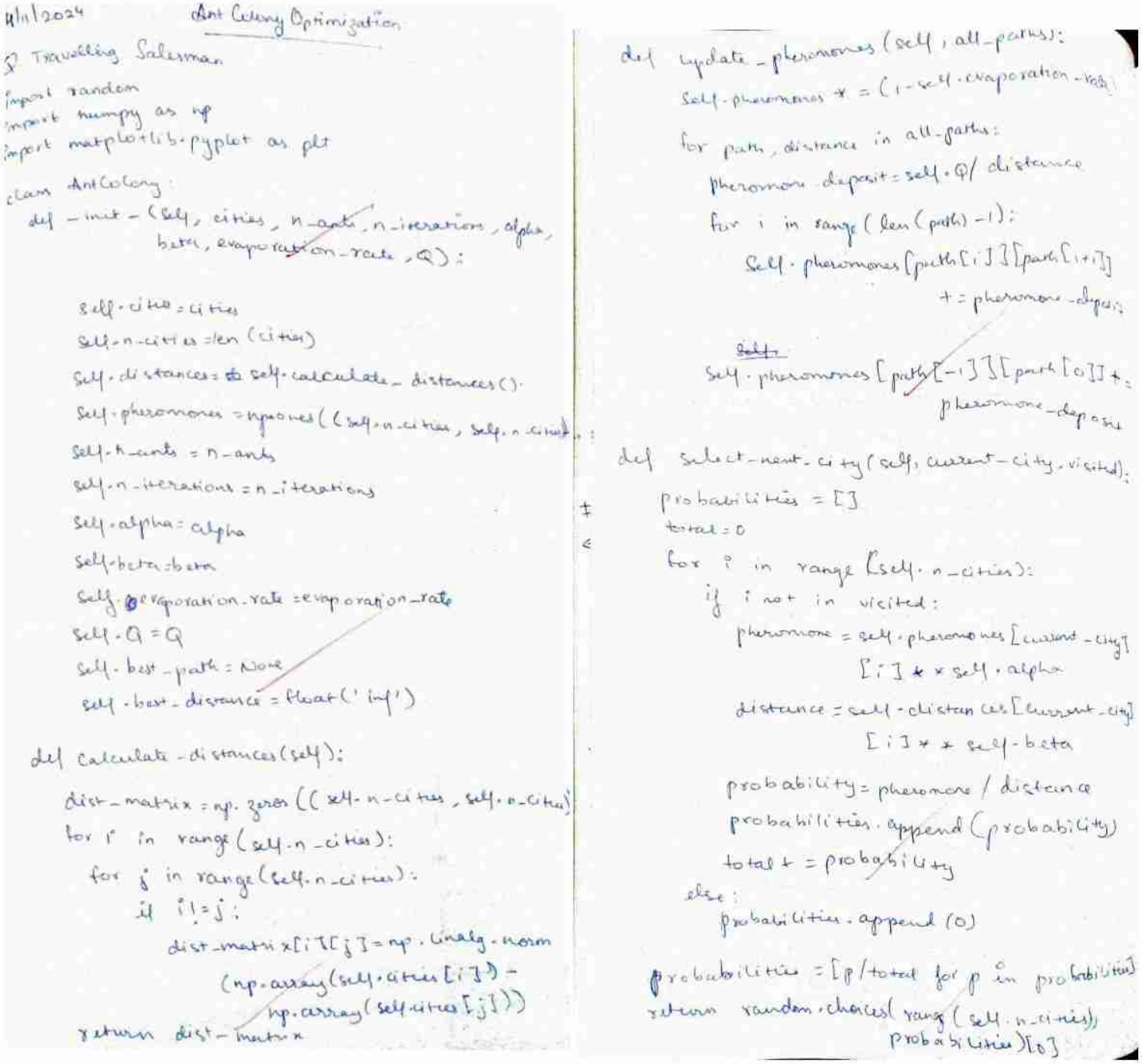
Iteration 10: Global Best Value = [100.]
Iteration 11: Global Best Value = [100.]
Iteration 12: Global Best Value = [100.]
Iteration 13: Global Best Value = [100.]
Iteration 14: Global Best Value = [100.]
Iteration 15: Global Best Value = [100.]
.
.
.
Iteration 98: Global Best Value = [100.]
Iteration 99: Global Best Value = [100.]
Iteration 100: Global Best Value = [100.]

Global best position: [-10.]
Global best value: [100.]

Program 3

Ant Colony Optimization for the Traveling Salesman Problem: The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:



```
4/11/2024                Ant Colony Optimization
# Travelling Salesman
import random
import numpy as np
import matplotlib.pyplot as plt

class AntColony:
    def __init__(self, cities, n_ants, n_iterations, alpha,
                 beta, evaporation_rate, Q):

        self.cities = cities
        self.n_cities = len(cities)
        self.distances = self.calculate_distances()
        self.pheromones = np.ones((self.n_cities, self.n_cities))
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.Q = Q
        self.best_path = None
        self.best_distance = float('inf')

    def calculate_distances(self):
        dist_matrix = np.zeros((self.n_cities, self.n_cities))
        for i in range(self.n_cities):
            for j in range(self.n_cities):
                if i != j:
                    dist_matrix[i][j] = np.linalg.norm
                        (np.array(self.cities[i]) -
                         np.array(self.cities[j]))
        return dist_matrix

    def update_pheromones(self, all_paths):
        self.pheromones *= (1 - self.evaporation_rate)
        for path, distance in all_paths:
            pheromone_deposit = self.Q / distance
            for i in range(len(path) - 1):
                self.pheromones[path[i]][path[i+1]]
                    += pheromone_deposit
            self.pheromones[path[-1]][path[0]] +=
                pheromone_deposit

    def select_next_city(self, current_city, visited):
        probabilities = []
        total = 0
        for i in range(self.n_cities):
            if i not in visited:
                pheromone = self.pheromones[current_city]
                    [i] ** self.alpha
                distance = self.distances[current_city]
                    [i] ** self.beta
                probability = pheromone / distance
                probabilities.append(probability)
                total += probability
            else:
                probabilities.append(0)
        probabilities = [p/total for p in probabilities]
        return random.choices(range(self.n_cities),
                              probabilities)[0]
```

```
def run(self):
    for iteration in range(self.n_iterations):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.construct_solution()
            distance = self.calculate_path_distance(path)
            all_paths.append((path, distance))

            if distance < self.best_distance:
                self.best_distance = distance
                self.best_path = path

        self.update_pheromones(all_paths)
        print(f"Iteration {iteration + 1}/{self.n_iterations}:
              Best Distance = {self.best_distance}")
    return self.best_path, self.best_distance

def construct_solution(self):
    path = []
    visited = set()
    current_city = random.randint(0, self.n_cities-1)
    path.append(current_city)
    visited.add(current_city)
    for _ in range(self.n_cities-1):
        next_city = self.select_next_city(current_city, visited)
        path.append(next_city)
        visited.add(next_city)
        current_city = next_city
    return path

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

if __name__ == "__main__":
    cities = [(0,0), (1,3), (4,3), (6,1), (3,0),
              (5,4), (7,6), (8,3)]

    aco = AntColony(
        cities=cities,
        n_ants=20,
        n_iterations=100,
        alpha=1,
        beta=2,
        evaporation_rate=0.5
        Q=100)

    best_path, best_distance = aco.run()
    print("Best Path:", best_path)
    print("Best Distance:", best_distance)
```

aco.plot_best_path()

Output:

Iteration 1/100 ; Best Distance = 22.557907

Iteration 2/100 : Best Distance = 22.557907

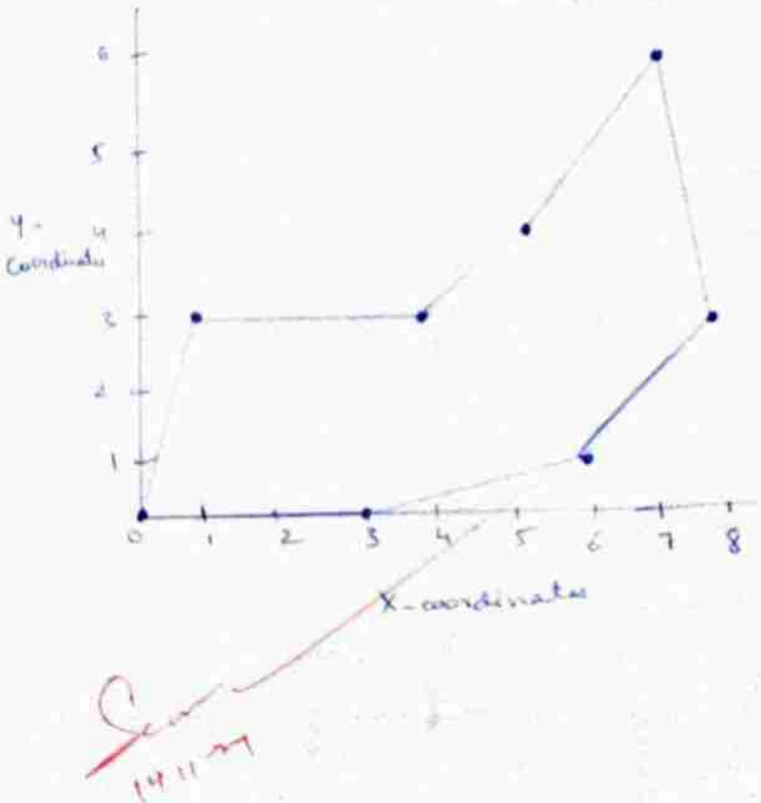
:

:

Iteration 100/100 : Best Distance = 22.557907

Best Path : [2, 5, 6, 7, 3, 4, 0, 1]

Best Distance : 22.557907



14/11/24

Code:

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

class AntColony:
    def __init__(self, cities, n_ants, n_iterations, alpha, beta, evaporation_rate, Q):
        """
        Initialize the Ant Colony Optimization parameters.
        cities: List of coordinates for the cities.
        n_ants: Number of ants in each iteration.
        n_iterations: Number of iterations.
        alpha: Influence of pheromone on path selection.
        beta: Influence of distance on path selection.
        evaporation_rate: Rate at which pheromone evaporates.
        Q: Constant used to calculate pheromone updates.
        """
        self.cities = cities
        self.n_cities = len(cities)
        self.distances = self.calculate_distances()
        self.pheromones = np.ones((self.n_cities, self.n_cities))  # Initial pheromone levels
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.Q = Q
        self.best_path = None
        self.best_distance = float('inf')

    def calculate_distances(self):
        """Calculate the distance matrix between all cities."""
        dist_matrix = np.zeros((self.n_cities, self.n_cities))
        for i in range(self.n_cities):
            for j in range(self.n_cities):
                if i != j:
                    dist_matrix[i][j] = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
        return dist_matrix

    def update_pheromones(self, all_paths):
        """Update the pheromones based on the paths found by the ants."""
        # Evaporate pheromones
        self.pheromones *= (1 - self.evaporation_rate)
```

```
    # Add new pheromones based on the paths found by the ants
    for path, distance in all_paths:
        pheromone_deposit = self.Q / distance
        for i in range(len(path) - 1):
            self.pheromones[path[i]][path[i+1]] += pheromone_deposit
        self.pheromones[path[-1]][path[0]] += pheromone_deposit  # Return to start

def select_next_city(self, current_city, visited):
    """Select the next city based on pheromone levels and distance."""
    probabilities = []
    total = 0
    for i in range(self.n_cities):
        if i not in visited:
            pheromone = self.pheromones[current_city][i] ** self.alpha
            distance = self.distances[current_city][i] ** self.beta
            probability = pheromone / distance
            probabilities.append(probability)
            total += probability
        else:
            probabilities.append(0)

    # Normalize probabilities
    probabilities = [p / total for p in probabilities]
    return random.choices(range(self.n_cities), probabilities)[0]

def run(self):
    """Run the ACO algorithm."""
    for iteration in range(self.n_iterations):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.construct_solution()
            distance = self.calculate_path_distance(path)
            all_paths.append((path, distance))

            # Update the best path found
            if distance < self.best_distance:
                self.best_distance = distance
                self.best_path = path

        # Update pheromones after all ants have completed their tour
        self.update_pheromones(all_paths)

        print(f"Iteration {iteration+1}/{self.n_iterations}: Best Distance =
```

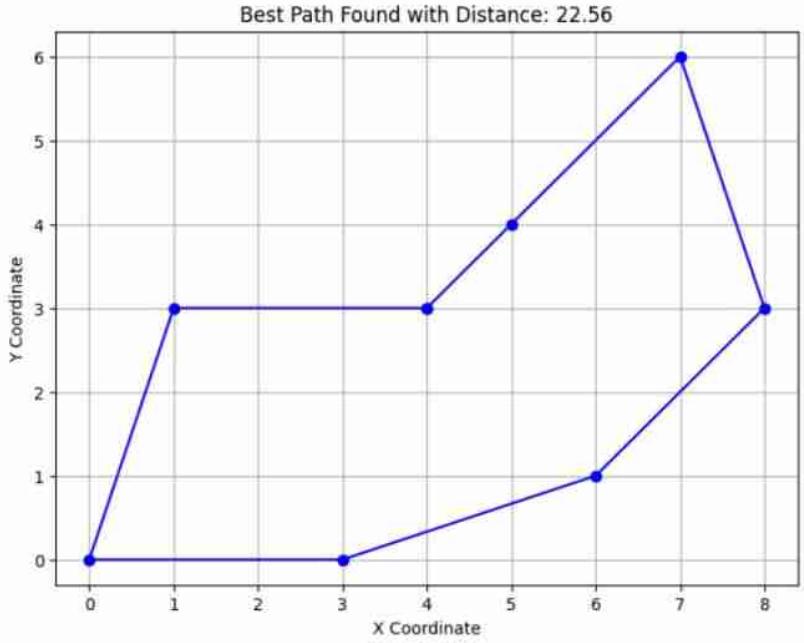
```
{self.best_distance}")

        return self.best_path, self.best_distance

    def construct_solution(self):
        """Construct a solution by letting an ant move from city to city."""
        path = []
        visited = set()
        current_city = random.randint(0, self.n_cities - 1)
        path.append(current_city)
        visited.add(current_city)

        for _ in range(self.n_cities - 1):
            next_city = self.select_next_city(current_city, visited)
            path.append(next_city)
            visited.add(next_city)
            current_city = next_city

        return path

    def calculate_path_distance(self, path):
        """Calculate the total distance of the path."""
        distance = 0
        for i in range(len(path) - 1):
            distance += self.distances[path[i]][path[i+1]]
        distance += self.distances[path[-1]][path[0]]  # Return to start
        return distance

    def plot_best_path(self):
        """Visualize the best path found."""
        if self.best_path is None:
            print("No path found yet.")
            return

        best_path_coords = [self.cities[i] for i in self.best_path] + [self.cities[self.best_path[0]]]
        x, y = zip(*best_path_coords)

        plt.figure(figsize=(8, 6))
        plt.plot(x, y, marker='o', color='b')
        plt.scatter(x, y, color='r')
        plt.title(f"Best Path Found with Distance: {self.best_distance:.2f}")
        plt.xlabel("X Coordinate")
        plt.ylabel("Y Coordinate")
        plt.grid(True)
```

```
        plt.show()

# Example usage
if __name__ == "__main__":
    # Example list of cities (coordinates)
    cities = [
        (0, 0), (1, 3), (4, 3), (6, 1), (3, 0), (5, 4), (7, 6), (8, 3)
    ]

    # Initialize the ACO
    aco = AntColony(
        cities=cities,
        n_ants=20,
        n_iterations=100,
        alpha=1,       # Pheromone influence
        beta=2,        # Distance influence
        evaporation_rate=0.5,
        Q=100
    )

    # Run ACO
    best_path, best_distance = aco.run()

    # Print the result
    print("Best Path:", best_path)
    print("Best Distance:", best_distance)

    # Plot the best path
    aco.plot_best_path()
```

Output:

```
Iteration 1/100: Best Distance = 22.557900792370614
Iteration 2/100: Best Distance = 22.557900792370614
Iteration 3/100: Best Distance = 22.557900792370614
Iteration 4/100: Best Distance = 22.557900792370614
Iteration 5/100: Best Distance = 22.557900792370614
Iteration 6/100: Best Distance = 22.557900792370614
Iteration 7/100: Best Distance = 22.557900792370614
Iteration 8/100: Best Distance = 22.557900792370614
Iteration 9/100: Best Distance = 22.557900792370614
Iteration 10/100: Best Distance = 22.557900792370614
Iteration 11/100: Best Distance = 22.557900792370614
Iteration 12/100: Best Distance = 22.557900792370614
```

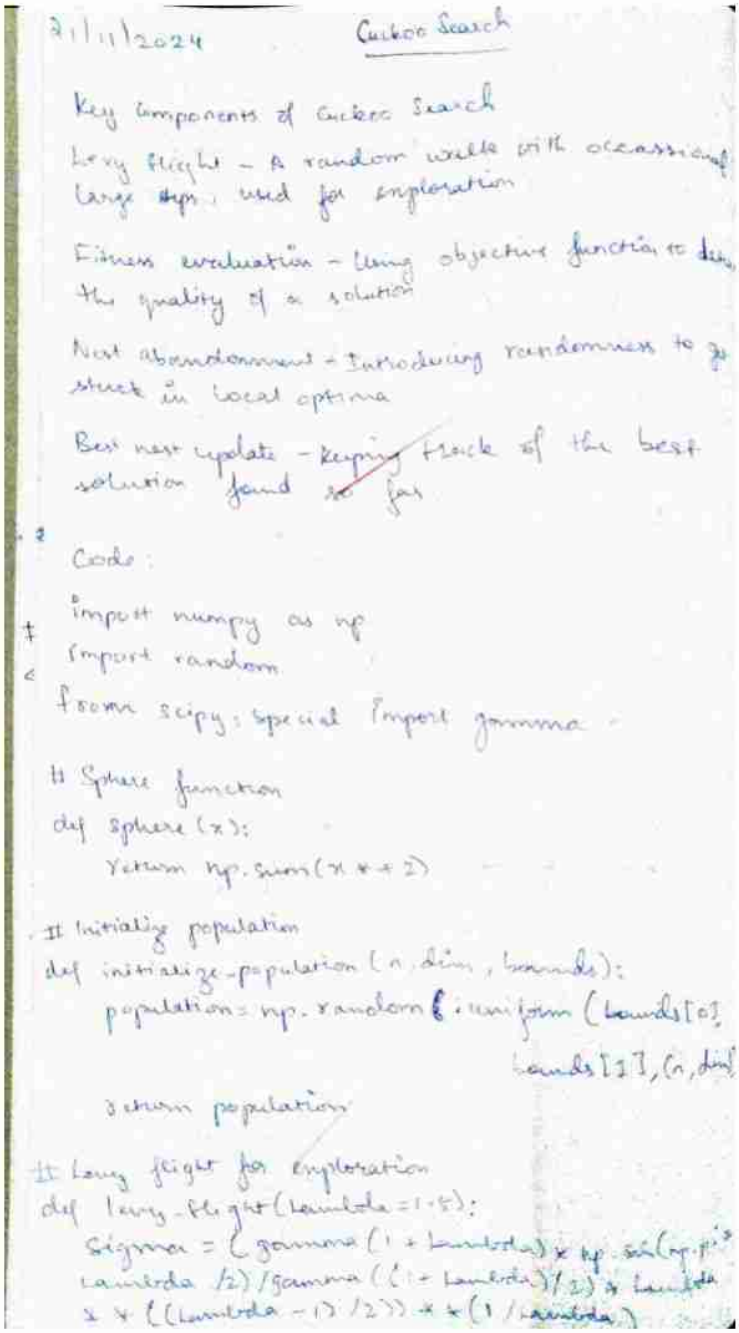
Iteration 13/100: Best Distance = 22.557900792370614
Iteration 14/100: Best Distance = 22.557900792370614
Iteration 15/100: Best Distance = 22.557900792370614
.
.
.
Iteration 98/100: Best Distance = 22.557900792370614
Iteration 99/100: Best Distance = 22.557900792370614
Iteration 100/100: Best Distance = 22.557900792370614
Best Path: [2, 5, 6, 7, 3, 4, 0, 1]
Best Distance: 22.557900792370614



Program 4

Cuckoo Search (CS): Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior 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:



21/11/2024 Cuckoo Search

Key components of Cuckoo Search

Levy flight - A random walk with occassional large steps used for exploration

Fitness evaluation - Using objective function to determine the quality of a solution

Nest abandonment - Introducing randomness to get stuck in local optima

Best nest update - Keeping track of the best solution found so far

Code:

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

# Sphere function
def sphere(x):
    return np.sum(x**2)

# Initialize population
def initialize_population(n, dim, bounds):
    population = np.random.uniform(bounds[0],
                                   bounds[1],(n,dim)
    return population

# Levy flight for exploration
def levy_flight(lambda=1.5):
    sigma = (gamma(1 + lambda) * np.sin(np.pi*
    lambda/2)/gamma((1+lambda)/2) * lambda
    2 ** ((lambda-1)/2)) ** (1/lambda)
```

Code:

```
import numpy as np
import random
from scipy.special import gamma  # Import the gamma function

# Sphere Function to optimize
def sphere(x):
    return np.sum(x**2)

# Initialize population
def initialize_population(n, dim, bounds):
    population = np.random.uniform(bounds[0], bounds[1], (n, dim))
    return population

# Lévy flight for exploration
def levy_flight(Lambda=1.5):
    sigma = (gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) / gamma((1 + Lambda) / 2) *
Lambda ** ((Lambda - 1) / 2)) ** (1 / Lambda)  # Use gamma instead of Gamma
    u = np.random.normal(0, sigma, size=1)
    v = np.random.normal(0, 1, size=1)
    step = u / (np.abs(v) ** (1 / Lambda))
    return step

# Cuckoo Search Algorithm
def cuckoo_search(func, n=50, dim=5, max_iter=1000, bounds=(-5.12, 5.12), pa=0.25):
    # Initialize nests (population)
    nests = initialize_population(n, dim, bounds)

    # Evaluate fitness of the population
    fitness = np.array([func(nest) for nest in nests])

    # Best solution
    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]

    # Main loop
    for _ in range(max_iter):
        # Generate new solutions (cuckoo search)
        new_nests = np.copy(nests)

        for i in range(n):
            # Lévy flight for generating new solutions
```

```
            step = levy_flight()
            new_nests[i] = nests[i] + step * (nests[i] - best_nest)  # Move towards the best nest

            # Boundary control
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])

            # Evaluate new fitness
            new_fitness = func(new_nests[i])

            # If new solution is better, replace it
            if new_fitness < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness

        # Abandon some nests (based on the probability pa)
        for i in range(n):
            if random.random() < pa:
                nests[i] = initialize_population(1, dim, bounds)[0]  # Reinitialize with new random
solution
                fitness[i] = func(nests[i])

        # Update the best solution
        best_idx = np.argmin(fitness)
        best_nest = nests[best_idx]
        best_fitness = fitness[best_idx]

    return best_nest, best_fitness

# Main execution
if __name__ == "__main__":
    best_solution, best_value = cuckoo_search(sphere, n=50, dim=10, max_iter=1000,
bounds=(-5.12, 5.12), pa=0.25)
    print("Best Solution: ", best_solution)
    print("Best Fitness (Value of Sphere Function): ", best_value)
```

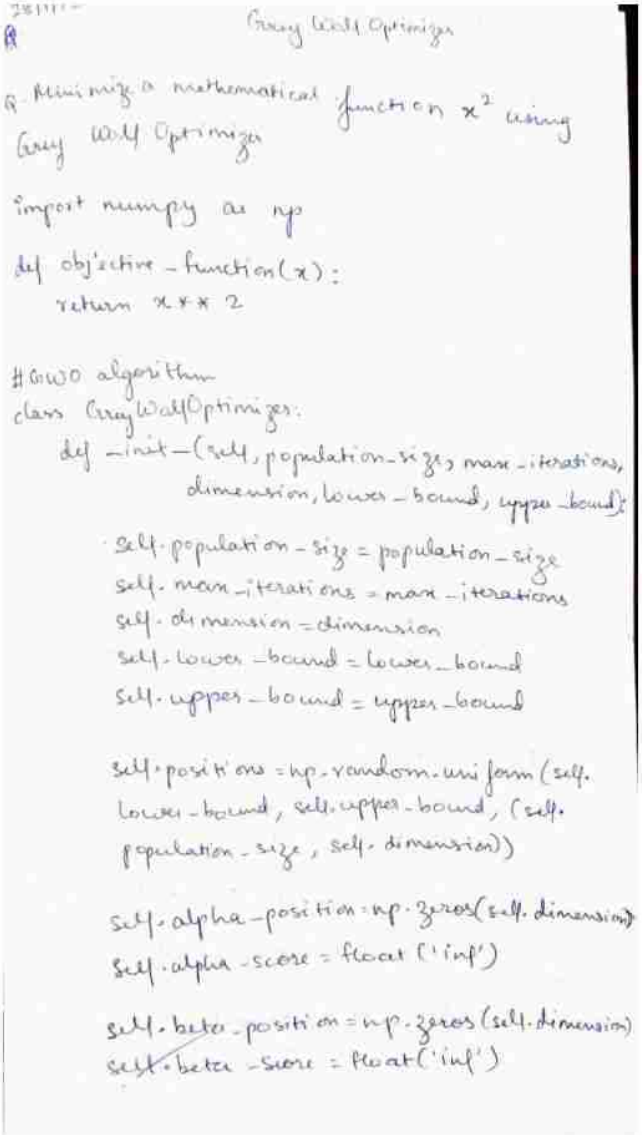
Output:

```
Best Solution:  [-0.02122667  0.03639467 -0.06056118 -0.02738857 -0.00015126
0.04255517
 -0.06440598 -0.01718141  0.00703945  0.00599116]
Best Fitness (Value of Sphere Function):  0.012532678828215921
```

Program 5

Grey Wolf Optimizer (GWO): The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior 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.

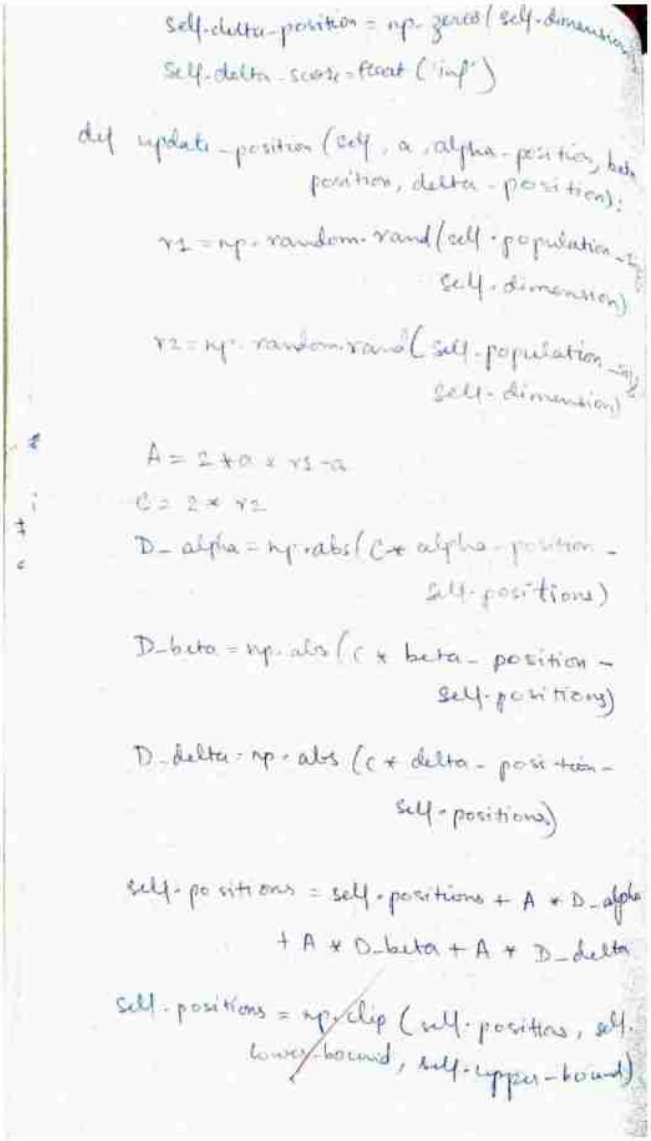
Algorithm:



Grey Wolf Optimizer

Q. Minimize a mathematical function x^2 using Grey Wolf Optimizer

```
import numpy as np

def objective_function(x):
    return x ** 2

#GWO algorithm
class GreyWolfOptimizer:
    def __init__(self, population_size, max_iterations,
                 dimension, lower_bound, upper_bound):
        self.population_size = population_size
        self.max_iterations = max_iterations
        self.dimension = dimension
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound

        self.positions = np.random.uniform(self.
        lower_bound, self.upper_bound, (self.
        population_size, self.dimension))

        self.alpha_position = np.zeros(self.dimension)
        self.alpha_score = float('inf')

        self.beta_position = np.zeros(self.dimension)
        self.beta_score = float('inf')
```



```
        self.delta_position = np.zeros(self.dimension)
        self.delta_score = float('inf')

    def update_position(self, a, alpha_position, beta
                        position, delta_position):
        r1 = np.random.rand(self.population_size,
                            self.dimension)
        r2 = np.random.rand(self.population_size,
                            self.dimension)

        A = 2 * a * r1 - a
        C = 2 * r2
        D_alpha = np.abs(C * alpha_position -
                         self.positions)
        D_beta = np.abs(C * beta_position -
                        self.positions)
        D_delta = np.abs(C * delta_position -
                         self.positions)
        self.positions = self.positions + A * D_alpha
                         + A * D_beta + A * D_delta
        self.positions = np.clip(self.positions, self.
                         lower_bound, self.upper_bound)
```

```
def optimize(self):
    for t in range(self.max_iterations):
        a = 2 - t * (2 / self.max_iterations)
        for i in range(self.population_size):
            fitness = objective_function(self.positions[i])

            if fitness < self.alpha_score:
                self.alpha_score = fitness
                self.alpha_position = self.positions[i]
            elif fitness < self.beta_score:
                self.beta_score = fitness
                self.beta_position = self.positions[i]
            elif fitness < self.delta_score:
                self.delta_score = fitness
                self.delta_position = self.positions[i]
        self.update_position(a, self.alpha_position,
                             self.beta_position, self.delta_position)

    print(f"Iteration {t+1}/{self.max_iterations}")
    print(f"Best Solution: {self.alpha_position}")
    print(f"Best Fitness: {self.alpha_score}")
```

```
    return self.alpha_position, ...

population_size = 30
max_iterations = 100
dimension = 1
lower_bound = -10
upper_bound = 10

gwo = GreyWolfOptimizer(population_size, max_iterations, dimension, lower_bound,
                        upper_bound)

best_position, best_fitness = gwo.optimize()

print(f"Best Position: {best_position}, Best Fitness (Objective Function Value):
      {best_fitness}")
```

Output:

```
Iteration 1/100, Best Solution: [0.42461468],
                 Best Fitness: [0.18029762]

Iteration 2/100, Best Solution: [0.42461468],
                 Best Fitness: [0.18029762]
   ⋮
Iteration 100/100, Best Solution [-3.12047946e-08]
                   Best fitness [9.73739...e-16]

Best position: [-3.1204 7946e-08]
Best fitness: [9.73739...e-16]
```

Scan 28.11.2...

Code:

```
import numpy as np

# Objective function (Mathematical function to be optimized)
def objective_function(x):
    return x**2  # f(x) = x^2

# GWO Algorithm
class GreyWolfOptimizer:
    def __init__(self, population_size, max_iterations, dimension, lower_bound,
upper_bound):
        self.population_size = population_size  # Number of wolves (agents)
        self.max_iterations = max_iterations  # Max iterations (stopping criteria)
        self.dimension = dimension  # Dimension of the search space (for single variable
optimization, it's 1)
        self.lower_bound = lower_bound  # Lower bound of search space
        self.upper_bound = upper_bound  # Upper bound of search space

        # Initialize the positions of the wolves (randomly)
        self.positions = np.random.uniform(self.lower_bound, self.upper_bound,
(self.population_size, self.dimension))

        # Initialize alpha, beta, delta wolves (best three solutions)
        self.alpha_position = np.zeros(self.dimension)
        self.alpha_score = float('inf')

        self.beta_position = np.zeros(self.dimension)
        self.beta_score = float('inf')

        self.delta_position = np.zeros(self.dimension)
        self.delta_score = float('inf')

    def update_position(self, a, alpha_position, beta_position, delta_position):
        # Update the positions of the wolves based on the leadership hierarchy
        r1 = np.random.rand(self.population_size, self.dimension)
        r2 = np.random.rand(self.population_size, self.dimension)
        A = 2 * a * r1 - a
        C = 2 * r2
        D_alpha = np.abs(C * alpha_position - self.positions)
        D_beta = np.abs(C * beta_position - self.positions)
        D_delta = np.abs(C * delta_position - self.positions)
```

```
        # Update positions of the wolves
        self.positions = self.positions + A * D_alpha + A * D_beta + A * D_delta
        self.positions = np.clip(self.positions, self.lower_bound, self.upper_bound)

    def optimize(self):
        # GWO Optimization process
        for t in range(self.max_iterations):
            a = 2 - t * (2 / self.max_iterations)  # Decreasing coefficient

            # Evaluate the fitness of each wolf
            for i in range(self.population_size):
                fitness = objective_function(self.positions[i])

                # Update the alpha, beta, and delta wolves
                if fitness < self.alpha_score:
                    self.alpha_score = fitness
                    self.alpha_position = self.positions[i]
                elif fitness < self.beta_score:
                    self.beta_score = fitness
                    self.beta_position = self.positions[i]
                elif fitness < self.delta_score:
                    self.delta_score = fitness
                    self.delta_position = self.positions[i]

            # Update the position of all wolves based on alpha, beta, delta wolves
            self.update_position(a, self.alpha_position, self.beta_position, self.delta_position)

            # Print the current best solution at each iteration
            print(f"Iteration {t+1}/{self.max_iterations}, Best Solution: {self.alpha_position},
Best Fitness: {self.alpha_score}")

        return self.alpha_position, self.alpha_score

# Parameters
population_size = 30  # Number of wolves
max_iterations = 100  # Maximum number of iterations
dimension = 1  # We are optimizing a 1D function (x^2)
lower_bound = -10  # Lower bound of the search space
upper_bound = 10   # Upper bound of the search space

# Initialize the GWO and start optimization
gwo = GreyWolfOptimizer(population_size, max_iterations, dimension, lower_bound,
upper_bound)
best_position, best_fitness = gwo.optimize()
```

```
print(f"Best Position: {best_position}, Best Fitness (Objective Function Value):
{best_fitness}")
```

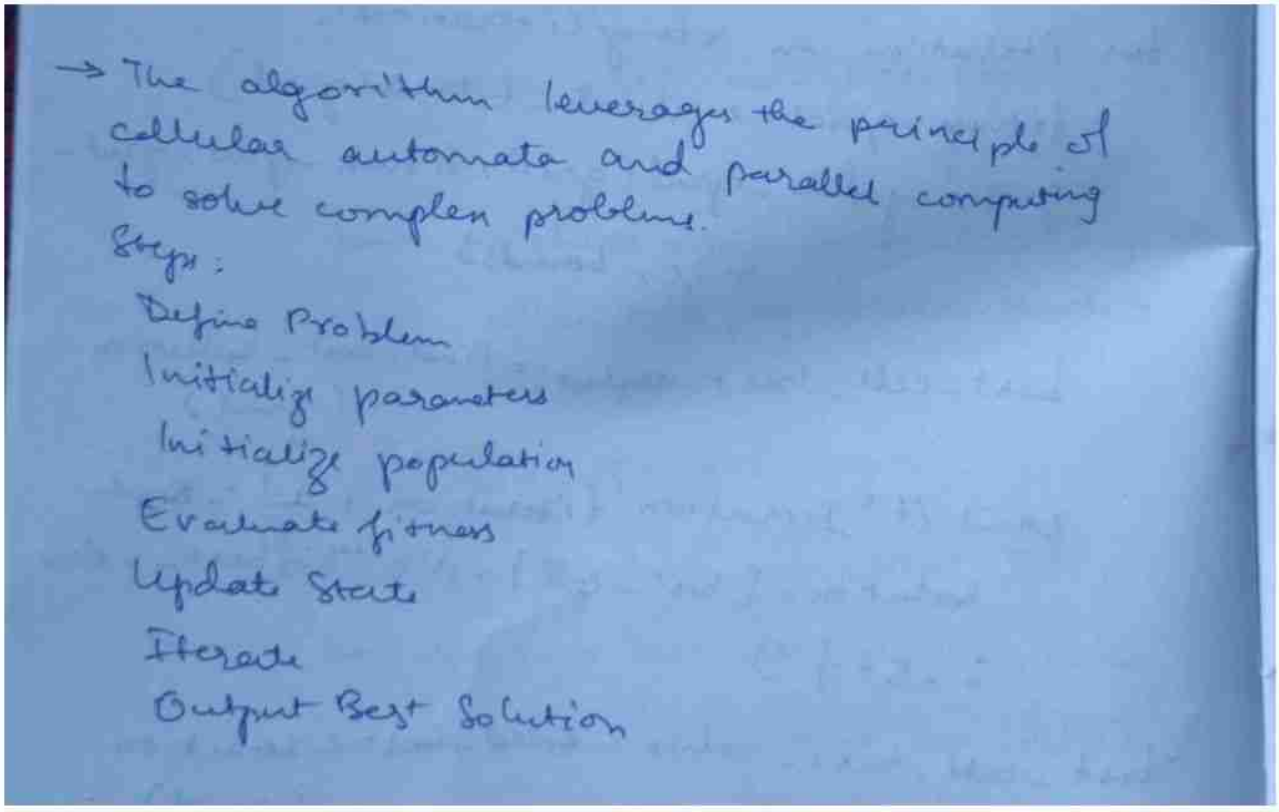
Output:

```
Iteration 1/100, Best Solution: [0.42461468], Best Fitness: [0.18029762]
Iteration 2/100, Best Solution: [0.42461468], Best Fitness: [0.18029762]
Iteration 3/100, Best Solution: [-0.25998381], Best Fitness: [0.06759158]
Iteration 4/100, Best Solution: [-0.25998381], Best Fitness: [0.06759158]
Iteration 5/100, Best Solution: [-0.25998381], Best Fitness: [0.06759158]
Iteration 6/100, Best Solution: [-0.14834999], Best Fitness: [0.02200772]
Iteration 7/100, Best Solution: [-0.14834999], Best Fitness: [0.02200772]
Iteration 8/100, Best Solution: [-0.14834999], Best Fitness: [0.02200772]
Iteration 9/100, Best Solution: [-0.14834999], Best Fitness: [0.02200772]
Iteration 10/100, Best Solution: [-0.14834999], Best Fitness: [0.02200772]
Iteration 11/100, Best Solution: [-0.14834999], Best Fitness: [0.02200772]
Iteration 12/100, Best Solution: [0.06708524], Best Fitness: [0.00450043]
Iteration 13/100, Best Solution: [0.06708524], Best Fitness: [0.00450043]
Iteration 14/100, Best Solution: [0.06708524], Best Fitness: [0.00450043]
Iteration 15/100, Best Solution: [0.06708524], Best Fitness: [0.00450043]
.
.
.
Iteration 98/100, Best Solution: [-3.1204794e-08], Best Fitness: [9.7373917e-16]
Iteration 99/100, Best Solution: [-3.1204794e-08], Best Fitness: [9.7373917e-16]
Iteration 100/100, Best Solution: [-3.1204794e-08], Best Fitness: [9.7373917e-16]
Best Position: [-3.1204794e-08], Best Fitness (Objective Function Value): [9.7373917e-16]
```

Program 6

Parallel Cellular Algorithms and Programs: Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:



Code:

```
import numpy as np
import random

def sphere_function(x, y):
    """
    Sphere function to be optimized.
```

```
    Minimum value at (0, 0) with f(0, 0) = 0
    """
    return x**2 + y**2

def initialize_grid(grid_size, bounds):
    """
    Initialize a grid of cells with random positions in the solution space.
    Each cell contains a tuple (x, y) representing its position.
    """
    grid = []
    for _ in range(grid_size):
        row = [(random.uniform(bounds[0], bounds[1]), random.uniform(bounds[0],
bounds[1]))
               for _ in range(grid_size)]
        grid.append(row)
    return grid

def evaluate_fitness(grid):
    """
    Evaluate the fitness of each cell in the grid.
    Fitness is calculated as the function value (lower is better).
    """
    fitness_grid = []
    for row in grid:
        fitness_row = [sphere_function(x, y) for x, y in row]
        fitness_grid.append(fitness_row)
    return fitness_grid

def get_neighbors(grid, i, j, grid_size):
    """
    Retrieve the neighbors of a cell at position (i, j).
    Uses a Moore neighborhood (8 neighbors).
    Handles edge wrapping for toroidal grid.
    """
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di == 0 and dj == 0:
                continue  # Skip the cell itself
            ni, nj = (i + di) % grid_size, (j + dj) % grid_size
            neighbors.append(grid[ni][nj])
    return neighbors

def update_grid(grid, fitness_grid, grid_size, bounds):
```

```
    """
    Update each cell's state based on its neighbors.
    The new state is influenced by the best neighbor.
    """
    new_grid = []
    for i in range(grid_size):
        new_row = []
        for j in range(grid_size):
            neighbors = get_neighbors(grid, i, j, grid_size)
            # Find the best neighbor based on fitness
            best_neighbor = min(neighbors, key=lambda pos: sphere_function(*pos))
            # Update cell towards the best neighbor (simple averaging step)
            x, y = grid[i][j]
            new_x = (x + best_neighbor[0]) / 2
            new_y = (y + best_neighbor[1]) / 2
            # Clamp to bounds
            new_x = np.clip(new_x, bounds[0], bounds[1])
            new_y = np.clip(new_y, bounds[0], bounds[1])
            new_row.append((new_x, new_y))
        new_grid.append(new_row)
    return new_grid

def find_best_solution(grid):
    """
    Find the best solution in the grid (minimum fitness value).
    """
    best_cell = min((cell for row in grid for cell in row),
                    key=lambda pos: sphere_function(*pos))
    best_value = sphere_function(*best_cell)
    return best_cell, best_value

def parallel_cellular_algorithm(grid_size=10, bounds=(-5, 5), iterations=50):
    """
    Main function to execute the Parallel Cellular Algorithm.
    """
    # Step 1: Initialize the grid
    grid = initialize_grid(grid_size, bounds)

    # Iterate and update grid
    for iteration in range(iterations):
        fitness_grid = evaluate_fitness(grid)
        grid = update_grid(grid, fitness_grid, grid_size, bounds)

        # Find current best solution
```

```
        best_cell, best_value = find_best_solution(grid)
        print(f"Iteration {iteration + 1}: Best Solution = {best_cell}, Value = {best_value:.5f}")

    # Output the best solution
    best_cell, best_value = find_best_solution(grid)
    print("\nFinal Best Solution:")
    print(f"Position: {best_cell}, Value: {best_value:.5f}")

if __name__ == "__main__":
    parallel_cellular_algorithm(grid_size=10, bounds=(-5, 5), iterations=50)
```

Output:

```
Iteration 1: Best Solution = (0.07549939820990836, 0.12152770082983855), Value =
0.02047
Iteration 2: Best Solution = (-0.041841040360995785, 0.007470020427892687), Value =
0.00181
Iteration 3: Best Solution = (0.06945024682785716, -0.08083078396730897), Value =
0.01136
Iteration 4: Best Solution = (0.023689864503236624, 0.010980953928815346), Value =
0.00068
Iteration 5: Best Solution = (0.023689864503236624, 0.010980953928815346), Value =
0.00068
Iteration 6: Best Solution = (0.00396925880950981, -0.01647144053430704), Value =
0.00029
Iteration 7: Best Solution = (-0.015314684073954067, 0.005979926888358321), Value =
0.00027
Iteration 8: Best Solution = (7.034570834474167e-05, -0.0015399028327062123), Value =
0.00000
Iteration 9: Best Solution = (7.034570834474167e-05, -0.0015399028327062123), Value =
0.00000
Iteration 10: Best Solution = (0.0010280662719008653, -0.0003505462114161565), Value =
0.00000
Iteration 11: Best Solution = (-9.505349549632225e-05, 0.0014150499547291224), Value =
0.00000
Iteration 12: Best Solution = (0.000750881477320626, -0.0004806433346966702), Value =
0.00000
Iteration 13: Best Solution = (0.0003611660455062199, -0.00032479297238806044), Value
= 0.00000
Iteration 14: Best Solution = (0.00045325464704252584, -2.6375862517009343e-06), Value
= 0.00000
Iteration 15: Best Solution = (-0.00012995610634194767, -0.00014573984924198848),
Value = 0.00000
.
```

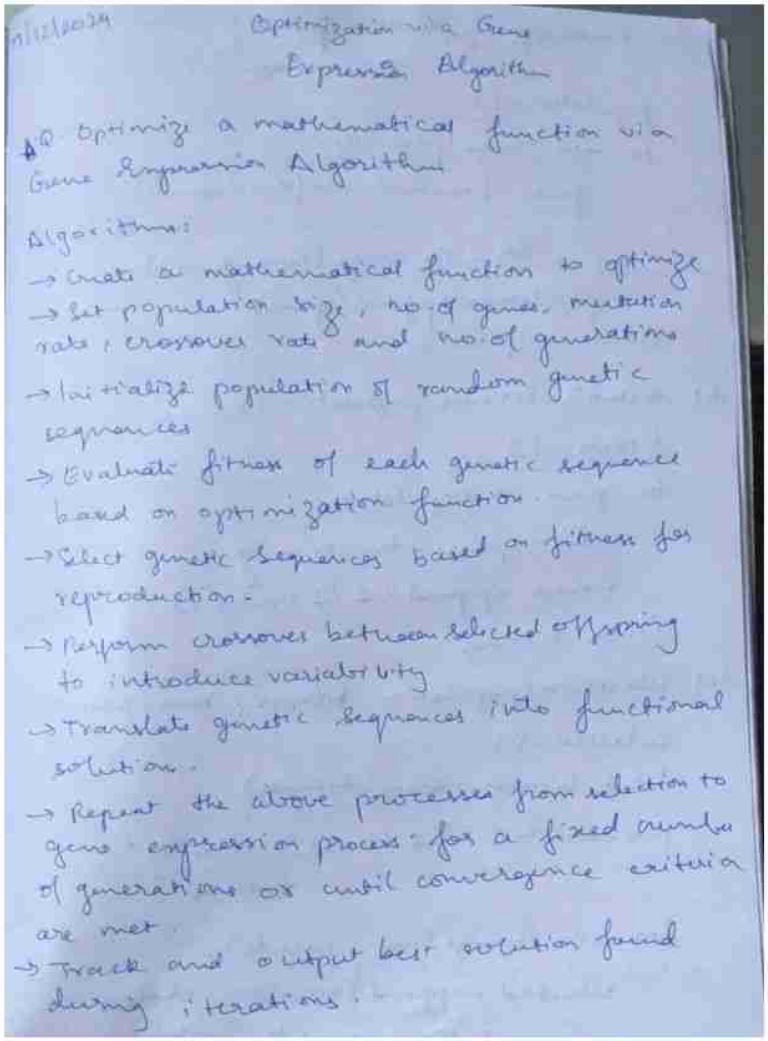
.
.
Iteration 46: Best Solution = (1.140623523845254e-05, -1.930517817029144e-06), Value = 0.00000
Iteration 47: Best Solution = (1.140623523845254e-05, -1.930517817029144e-06), Value = 0.00000
Iteration 48: Best Solution = (1.140623523845254e-05, -1.930517817029144e-06), Value = 0.00000
Iteration 49: Best Solution = (1.140623523845254e-05, -1.930517817029144e-06), Value = 0.00000
Iteration 50: Best Solution = (1.140623523845254e-05, -1.930517817029144e-06), Value = 0.00000

Final Best Solution:
Position: (1.140623523845254e-05, -1.930517817029144e-06), Value: 0.00000

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

Q. Optimize a mathematical function via Gene Expression Algorithm

Algorithm:

→ Create a mathematical function to optimize
→ Set population size, no of genes, mutation rate, crossover rate and no of generations
→ Initialize population of random genetic sequences
→ Evaluate fitness of each genetic sequence based on optimization function
→ Select genetic sequences based on fitness for reproduction.
→ Perform crossover between selected offspring to introduce variability
→ Translate genetic sequences into functional solutions.
→ Repeat the above processes from selection to gene expression process for a fixed number of generations or until convergence criteria are met.
→ Track and output best solution found during iterations.

Code:

```
import random
import numpy as np

def sphere_function(x):
    """
    Sphere function to be optimized.
    Minimum value at (0, 0, ..., 0) with f(0) = 0
    """
    return sum(xi ** 2 for xi in x)

def initialize_population(pop_size, num_genes, bounds):
    """
    Generate an initial population of random genetic sequences.
    """
    population = []
    for _ in range(pop_size):
        genes = [random.uniform(bounds[0], bounds[1]) for _ in range(num_genes)]
        population.append(genes)
    return population

def evaluate_fitness(population):
    """
    Evaluate the fitness of each genetic sequence in the population.
    Fitness is the inverse of the sphere function (minimization problem).
    """
    fitness = []
    for genes in population:
        value = sphere_function(genes)
        fitness.append(1 / (1 + value))  # Avoid division by zero
    return fitness

def selection(population, fitness, num_parents):
    """
    Select individuals based on their fitness using roulette wheel selection.
    """
    selected = []
    total_fitness = sum(fitness)
    probabilities = [f / total_fitness for f in fitness]
    for _ in range(num_parents):
        selected.append(random.choices(population, weights=probabilities, k=1)[0])
    return selected
```

```
def crossover(parents, num_offspring):
    """
    Perform crossover between pairs of parents to produce offspring.
    Single-point crossover is used.
    """
    offspring = []
    for _ in range(num_offspring):
        p1, p2 = random.sample(parents, 2)
        crossover_point = random.randint(1, len(p1) - 1)
        child = p1[:crossover_point] + p2[crossover_point:]
        offspring.append(child)
    return offspring

def mutation(offspring, mutation_rate, bounds):
    """
    Apply mutation to the offspring to introduce variability.
    Mutation randomly modifies genes based on the mutation rate.
    """
    for i in range(len(offspring)):
        for j in range(len(offspring[i])):
            if random.random() < mutation_rate:
                offspring[i][j] = random.uniform(bounds[0], bounds[1])
    return offspring

def gene_expression(population):
    """
    Gene expression step: Here, the genetic sequence itself acts as the solution.
    Returns the population as functional solutions.
    """
    return population

def find_best_solution(population):
    """
    Find the best solution in the population based on the sphere function.
    """
    best_solution = min(population, key=sphere_function)
    best_value = sphere_function(best_solution)
    return best_solution, best_value

def gene_expression_algorithm(pop_size=50, num_genes=10, bounds=(-5, 5),
mutation_rate=0.1,
                              crossover_rate=0.8, generations=100):
    """
    Main function to execute the Gene Expression Algorithm.
```

```
    """
    # Step 1: Initialize the population
    population = initialize_population(pop_size, num_genes, bounds)
    num_parents = int(pop_size * crossover_rate)
    num_offspring = pop_size - num_parents

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

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

        # Step 4: Crossover
        offspring = crossover(parents, num_offspring)

        # Step 5: Mutation
        offspring = mutation(offspring, mutation_rate, bounds)

        # Step 6: Gene Expression
        population = gene_expression(parents + offspring)

        # Step 7: Find the best solution in the current generation
        best_solution, best_value = find_best_solution(population)
        print(f"Generation {generation + 1}: Best Value = {best_value:.5f}")

    # Output the final best solution
    best_solution, best_value = find_best_solution(population)
    print("\nFinal Best Solution:")
    print(f"Genes: {best_solution}, Value: {best_value:.5f}")

if __name__ == "__main__":
    gene_expression_algorithm(pop_size=50, num_genes=10, bounds=(-5, 5),
mutation_rate=0.1,
                              crossover_rate=0.8, generations=500)
```

Output:

```
Generation 1: Best Value = 29.24526
Generation 2: Best Value = 29.24526
Generation 3: Best Value = 29.24526
Generation 4: Best Value = 20.07386
```

Generation 5: Best Value = 20.07386
Generation 6: Best Value = 20.07386
Generation 7: Best Value = 20.07386
Generation 8: Best Value = 20.07386
Generation 9: Best Value = 20.07386
Generation 10: Best Value = 20.07386
Generation 11: Best Value = 20.07386
Generation 12: Best Value = 4.47912
Generation 13: Best Value = 4.47912
Generation 14: Best Value = 3.81217
Generation 15: Best Value = 3.81217
.
.
.
Generation 495: Best Value = 0.34531
Generation 496: Best Value = 0.34531
Generation 497: Best Value = 0.34531
Generation 498: Best Value = 0.34531
Generation 499: Best Value = 0.34531
Generation 500: Best Value = 0.34531

Final Best Solution:
Genes: [0.37075091924379766, -0.23653661806836546, 0.020809139240634877, 0.2205485795065476, -0.21483044755642666, -0.04751087900227624, -0.12182781417202904, 0.045605826740818145, 0.03182324378767376, 0.1910008455308967], Value: 0.34531