

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sanketh M Hanasi(1BM22CS242)

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 **Sanketh M Hanasi (1BM22CS242)**, 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.

| | |
|--|--|
| Sonika Sharma D Assistant Professor Department of CSE, BMSCE | Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE |
|--|--|

Index

| Sl. No. | Date | Experiment Title | Page No. |
|--------------------|-------------|--|-----------------|
| 1 | 24/10/24 | Genetic Algorithm for Optimization Problems | 1 |
| 2 | 7/11/24 | Particle Swarm Optimization for Function Optimization | 5 |
| 3 | 14/11/24 | Ant Colony Optimization for the Traveling Salesman Problem | 9 |
| 4 | 21/11/24 | Cuckoo Search (CS) | 15 |
| 5 | 28/11/24 | Grey Wolf Optimizer (GWO) | 20 |
| 6 | 18/12/24 | Parallel Cellular Algorithms and Programs | 26 |
| 7 | 18/12/24 | Parallel Cellular Algorithms and Programs | 31 |

Github Link:

https://github.com/SankethHanasi/BIS_LAB

Program 1: Genetic Algorithm for Optimization on 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:

24/10/24

```
1) Genetic Algorithm for Optimization Problems:  
function GeneticAlgorithm():  
    initialize parameters (population size, mutation rate,  
    crossover rate, numgenerations, range_min, range_max)  
    population = InitializePopulation (population size,  
    range_min, range_max)  
  
    for generation in 1 to num_generations:  
        fitness = EvaluateFitness (population)  
        new_population = [Initial step 2]  
  
        for i from 1 to population_size / 2:  
            parent1, parent2 = Selection (population, fitness)  
            offspring1, offspring2 = Crossover (parent1, parent2)  
            new_population.append (Mutate (offspring1))  
            new_population.append (Mutate (offspring2))  
  
        new_population = new_population [Initial step 3]  
        best_fitness = Max(fitness)  
        best_solution = population [ArgMax (fitness)]  
  
    return best_solution, FitnessFunction (best_solution)
```

```

function fitness_function(x):
    return x**2

function initialize_population(size, range_min, range_max):
    return [random(range_min, range_max) for each i
            in range(size)]

function selection(population, fitness):
    selection_probs = []
    total_fitness = sum(fitness)
    for individual in population:
        selection_probs.append(individual.fitness / total_fitness)
    return selection_probs

probability = fitness / total_fitness
parent1 = select.individual from population based on selection
parent2 = select.individual from population based on selection
return parent1, parent2

function crossover(parent1, parent2):
    if random value < crossoverRate:
        crossoverPoint = random in btw 1 and chromosomeLength
        child1 = parent1 up to crossoverPoint + parent2 after crossover
        child2 = parent2 up to crossoverPoint + parent1 after crossover
    Else:
        child1, child2 = Parent1, Parent2
    return child1, child2

Function mutate(chromosome):
    for each bit in chromosome:
        if random value < mutationRate:
            flip the bit
    return chromosome

```

Code:

```
import numpy as np
import random
```

```
# Objective function to maximize
def objective_function(x):
    return x ** 2
```

```
# Initialize parameters
population_size = 100
num_generations = 50
```

```

mutation_rate = 0.1
crossover_rate = 0.7
range_min = -10
range_max = 10

# Create initial population
def initialize_population(size, min_val, max_val):
    return np.random.uniform(min_val, max_val, size)

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

# Selection using roulette-wheel method
def selection(population, fitness):
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    return population[np.random.choice(range(len(population)), size=2, p=probabilities)]

# Crossover between two parents
def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        return (parent1 + parent2) / 2 # Simple averaging for crossover
    return parent1 # No crossover

# Mutation of an individual
def mutate(individual):
    if random.random() < mutation_rate:
        return np.random.uniform(range_min, range_max)
    return individual

# Genetic Algorithm function
def genetic_algorithm():
    # Step 1: Initialize population
    population = initialize_population(population_size, range_min, range_max)

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

        # Track the best solution
        best_index = np.argmax(fitness)
        best_solution = population[best_index]
        best_fitness = fitness[best_index]

        # print(f"Generation {generation + 1}: Best Solution = {best_solution}, Fitness = {best_fitness}")

        # Step 3: Create new population

```

```

new_population = []
for _ in range(population_size):
    # Select parents
    parent1, parent2 = selection(population, fitness)
    # Crossover to create offspring
    offspring = crossover(parent1, parent2)
    # Mutate offspring
    offspring = mutate(offspring)
    new_population.append(offspring)

# Step 6: Replace old population with new population
population = np.array(new_population)

return best_solution, best_fitness

# Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm()
print(f"Best Solution Found: {best_solution}, Fitness: {best_fitness}")

```

OUTPUT:



Best Solution Found: -9.290037411642935, Fitness: 86.30479510972536

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:

7/11/24

2) Particle Swarm Optimization:

```
FUNCTION Rastrigin(x):
    return 10 * length(x) + sum (x[i]^2 - 10 * cos(2 * pi * x[i]) for
    each x[i] in x)
```

Class PSO:

```
Function __init__(objective_function, n_particles, n_dimensions,
max_iter, bounds):
    Set self.objective_function = objective_function
    Set self.n_particles = n_particles
    Set self.n_dimensions = n_dimensions
    Set self.max_iter = max_iter
    Set self.bounds = bounds
    Initialize self.positions as random positions in
    bounds for n_particles x n_dimensions
    Initialize self.velocities as random velocities for
    n_particles x n_dimensions
    Initialize self.pbest_positions as self.positions
    Initialize self.pbest_values as the fitness of self.positions
    using objective function

    Set self.gbest_position = position with the min. fitness
    Set self.gbest_value = min fitness from pbest values
```

Function update_velocity(i):

```
Randomly generate r1, r2 (uniform random numbers in
range [0,1])
inertia = self.w * self.velocities[i]
cognitive = self.c1 * r1 * (self.pbest_positions[i] - self.positions[i])
social = self.c2 * r2 * (self.gbest_position - self.positions[i])
Set self.velocities[i] = inertia + cognitive + social
```

Function update_position(i):
 Update position: self.positions[i] += self.velocities[i]
 self.positions[i] = np.clip(self.positions[i], self.bound_low[i],
 self.bound_high[i])

Function optimize():
 for t=1 to max_iter:
 for i=1 to n_particles:
 self.update_velocity(i)
 self.update_position(i)

fitness = self.objective_function(self.positions)

if fitness < self.pbest_values[i]:
 self.pbest_values[i] = fitness
 self.pbest_positions[i] = self.positions[i]

min_pbest_value = min value from pbest_values
 if min_pbest_value < self.gbest_value:
 set self.gbest_value = min_pbest_value
 set self.gbest_position = self.positions[i]

return self.gbest_position, self.gbest_value


 7/11/24

Code:

```

import numpy as np
import random

# Define the optimization problem (Rastrigin Function)
def rastrigin(x):
  A = 10
  return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Particle Swarm Optimization (PSO) implementation
class Particle:
  
```

```

def __init__(self, dimension, lower_bound, upper_bound):
    # Initialize the particle position and velocity randomly
    self.position = np.random.uniform(lower_bound, upper_bound, dimension)
    self.velocity = np.random.uniform(-1, 1, dimension)
    self.best_position = np.copy(self.position)
    self.best_value = rastrigin(self.position)

def update_velocity(self, global_best_position, w, c1, c2):
    # Update the velocity of the particle
    r1 = np.random.rand(len(self.position))
    r2 = np.random.rand(len(self.position))

    # Inertia term
    inertia = w * self.velocity

    # Cognitive term (individual best)
    cognitive = c1 * r1 * (self.best_position - self.position)

    # Social term (global best)
    social = c2 * r2 * (global_best_position - self.position)

    # Update velocity
    self.velocity = inertia + cognitive + social

def update_position(self, lower_bound, upper_bound):
    # Update the position of the particle
    self.position = self.position + self.velocity

    # Ensure the particle stays within the bounds
    self.position = np.clip(self.position, lower_bound, upper_bound)

def evaluate(self):
    # Evaluate the fitness of the particle
    fitness = rastrigin(self.position)

    # Update the particle's best position if necessary
    if fitness < self.best_value:
        self.best_value = fitness
        self.best_position = np.copy(self.position)

def particle_swarm_optimization(dim, lower_bound, upper_bound, num_particles=30, max_iter=100,
w=0.5, c1=1.5, c2=1.5):
    # Initialize particles
    particles = [Particle(dim, lower_bound, upper_bound) for _ in range(num_particles)]

    # Initialize the global best position and value
    global_best_position = particles[0].best_position
    global_best_value = particles[0].best_value
    for i in range(max_iter):

```

```

# Update each particle
for particle in particles:
    particle.update_velocity(global_best_position, w, c1, c2)
    particle.update_position(lower_bound, upper_bound)
    particle.evaluate()

    # Update global best position if needed
    if particle.best_value < global_best_value:
        global_best_value = particle.best_value
        global_best_position = np.copy(particle.best_position)

    # Optionally print the progress
    if (i+1) % 10 == 0:
        print(f"Iteration {i+1}/{max_iter} - Best Fitness: {global_best_value}")

return global_best_position, global_best_value

dim = 2          # Number of dimensions for the function
lower_bound = -5.12 # Lower bound of the search space
upper_bound = 5.12 # Upper bound of the search space
num_particles = 30 # Number of particles in the swarm
max_iter = 100    # Number of iterations

# Run the PSO
best_position, best_value = particle_swarm_optimization(dim, lower_bound, upper_bound,
num_particles, max_iter)

# Output the best solution found
print("\nBest Solution Found:")
print("Position:", best_position)
print("Fitness:", best_value)

```

OUTPUT:

```

→ Iteration 10/100 - Best Fitness: 1.1103296669969005
Iteration 20/100 - Best Fitness: 0.020031338560627887
Iteration 30/100 - Best Fitness: 2.788695226740856e-06
Iteration 40/100 - Best Fitness: 1.0778596895022474e-06
Iteration 50/100 - Best Fitness: 6.450946443692374e-10
Iteration 60/100 - Best Fitness: 2.0463630789890885e-11
Iteration 70/100 - Best Fitness: 1.0658141036401503e-14
Iteration 80/100 - Best Fitness: 0.0
Iteration 90/100 - Best Fitness: 0.0
Iteration 100/100 - Best Fitness: 0.0

Best Solution Found:
Position: [-1.63024230e-09  1.14735681e-09]
Fitness: 0.0

```

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:

```
14/11/24
③ Ant colony optimization (ACO):
# initialize parameters
n_cities, n_ants, iterations ← initialize parameters (e.g.)
alpha, beta, rho, Q ← initialize parameters (fixed)
cities = [random.randint(0,100), random.randint(0,100)]
for i in range(num_cities)
    # distance & pheromone matrix
def dist(c1, c2):
    return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)

dist_matrix = [[dist(cities[i], cities[j]) for j in range(n_cities)]
               for i in range(n_cities)]
pheromone_matrix = [[0.0 for i in range(n_cities)] for j in range(n_cities)]
# best solution initialization
best_path, best_distance = None, float('inf')
# ACO loop
for _ in range(iterations):
    all_paths, all_distances = [], []
    # ant path construction
    for _ in range(n_ants):
        unvisited = list(range(n_cities))
        path = [curr_city := random.choice(unvisited)]
        unvisited.remove(curr_city)
        # probabilities
        while unvisited:
            prob = [pheromone_matrix[curr_city][i]**alpha
                    * (1.0 / dist_matrix[curr_city][i])**beta
                    for i in unvisited]
            # choose next city based on probability
```

total_prob = sum(prob)
 probd = [p / total_prob for p in prob]
 city = next_city = random.choices(unvisited, probd)[0]
 path.append(next_city)
 unvisited.remove(next_city)
 currcity = next_city
 path.append(path[0])
 total_dist = sum(dist_matrix[path[i]][path[i+1]] for i in range(n_cities - 1))
 all_paths.append(path)
 all_distances.append(total_distance)

if total_distance < best_distance:
 best_path, best_distance = path, total_dist

for i in range(n_cities):
 for j in range(n_cities):
 pheromone_matrix[i][j] *= (1 - rho)
 if [i, j] in zip(best_path, best_path[1:] + best_path[:1]):
 pheromone_matrix[i][j] += Q / best_distance

8/11/21
 14/11/21

Code:

```

#ant colony
import numpy as np
import matplotlib.pyplot as plt

# 1. Define the Problem: Create a set of cities with their coordinates
cities = np.array([
  [0, 0], # City 0
  [1, 5], # City 1
  [5, 1], # City 2
])
  
```

```

[6, 4], # City 3
[7, 8], # City 4
])

# Calculate the distance matrix between each pair of cities
def calculate_distances(cities):
    num_cities = len(cities)
    distances = np.zeros((num_cities, num_cities))

    for i in range(num_cities):
        for j in range(num_cities):
            distances[i][j] = np.linalg.norm(cities[i] - cities[j])

    return distances

distances = calculate_distances(cities)

# 2. Initialize Parameters
num_ants = 10
num_cities = len(cities)
alpha = 1.0 # Influence of pheromone
beta = 5.0 # Influence of heuristic (inverse distance)
rho = 0.5 # Evaporation rate
num_iterations = 30
initial_pheromone = 1.0

# Pheromone matrix initialization
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# 3. Heuristic information (Inverse of distance)
def heuristic(distances):
    with np.errstate(divide='ignore'): # Ignore division by zero
        return 1 / distances

eta = heuristic(distances)

# 4. Choose next city probabilistically based on pheromone and heuristic info
def choose_next_city(pheromone, eta, visited):
    probs = []
    for j in range(num_cities):
        if j not in visited:
            pheromone_ij = pheromone[visited[-1], j] ** alpha
            heuristic_ij = eta[visited[-1], j] ** beta
            probs.append(pheromone_ij * heuristic_ij)
        else:
            probs.append(0)
    probs = np.array(probs)
    return np.random.choice(range(num_cities), p=probs / probs.sum())

```

```

# Construct solution for a single ant
def construct_solution(pheromone, eta):
    tour = [np.random.randint(0, num_cities)]
    while len(tour) < num_cities:
        next_city = choose_next_city(pheromone, eta, tour)
        tour.append(next_city)
    return tour

# 5. Update pheromones after all ants have constructed their tours
def update_pheromones(pheromone, all_tours, distances, best_tour):
    pheromone *= (1 - rho) # Evaporate pheromones

    # Add pheromones for each ant's tour
    for tour in all_tours:
        tour_length = sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)])
        for i in range(-1, num_cities - 1):
            pheromone[tour[i], tour[i + 1]] += 1.0 / tour_length

    # Increase pheromones on the best tour
    best_length = sum([distances[best_tour[i], best_tour[i + 1]] for i in range(-1, num_cities - 1)])
    for i in range(-1, num_cities - 1):
        pheromone[best_tour[i], best_tour[i + 1]] += 1.0 / best_length

# 6. Main ACO Loop: Iterate over multiple iterations to find the best solution
def run_aco(distances, num_iterations):
    pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
    best_tour = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        all_tours = [construct_solution(pheromone, eta) for _ in range(num_ants)]
        all_lengths = [sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)]) for tour in all_tours]

        current_best_length = min(all_lengths)
        current_best_tour = all_tours[all_lengths.index(current_best_length)]

        if current_best_length < best_length:
            best_length = current_best_length
            best_tour = current_best_tour

        update_pheromones(pheromone, all_tours, distances, best_tour)

        print(f"Iteration {iteration + 1}, Best Length: {best_length}")

    return best_tour, best_length

# Run the ACO algorithm
best_tour, best_length = run_aco(distances, num_iterations)

```

```

# 7. Output the Best Solution
print(f"Best Tour: {best_tour}")
print(f"Best Tour Length: {best_length}")

# 8. Plot the Best Route
def plot_route(cities, best_tour):
    plt.figure(figsize=(8, 6))
    for i in range(len(cities)):
        plt.scatter(cities[i][0], cities[i][1], color='red')
        plt.text(cities[i][0], cities[i][1], f"City {i}", fontsize=12)

    # Plot the tour as lines connecting the cities
    tour_cities = np.array([cities[i] for i in best_tour] + [cities[best_tour[0]]]) # Complete the loop by
    returning to the start
    plt.plot(tour_cities[:, 0], tour_cities[:, 1], linestyle='-', marker='o', color='blue')

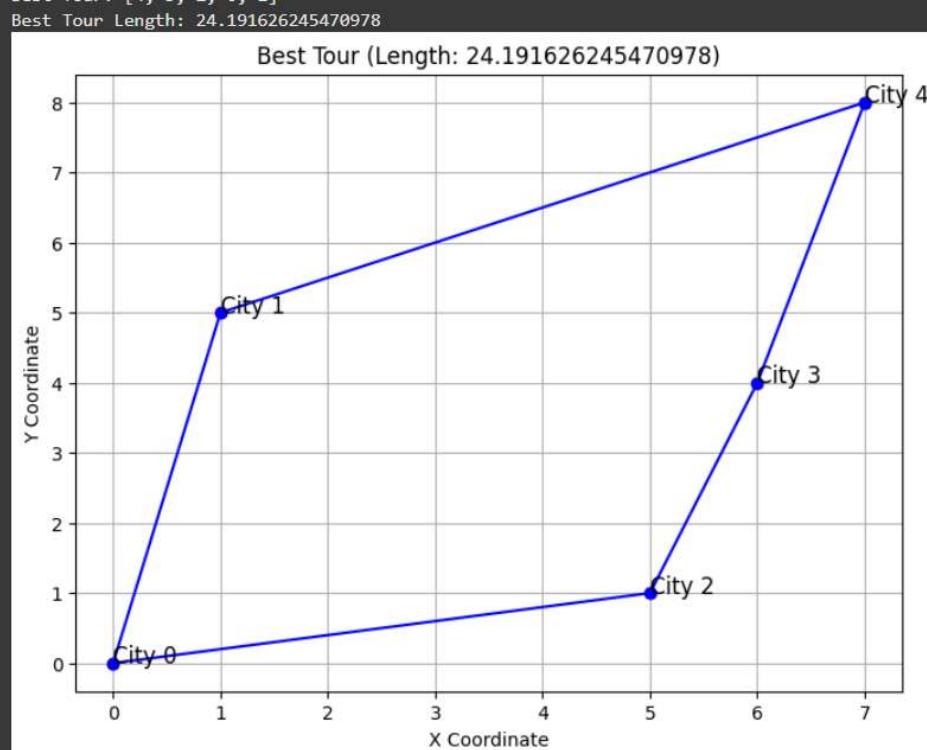
    plt.title(f"Best Tour (Length: {best_length})")
    plt.xlabel("X Coordinate")
    plt.ylabel("Y Coordinate")
    plt.grid(True)
    plt.show()

# Call the plot function
plot_route(cities, best_tour)

```

OUTPUT:

```
→ Iteration 1, Best Length: 24.191626245470978
Iteration 2, Best Length: 24.191626245470978
Iteration 3, Best Length: 24.191626245470978
Iteration 4, Best Length: 24.191626245470978
Iteration 5, Best Length: 24.191626245470978
Iteration 6, Best Length: 24.191626245470978
Iteration 7, Best Length: 24.191626245470978
Iteration 8, Best Length: 24.191626245470978
Iteration 9, Best Length: 24.191626245470978
Iteration 10, Best Length: 24.191626245470978
Iteration 11, Best Length: 24.191626245470978
Iteration 12, Best Length: 24.191626245470978
Iteration 13, Best Length: 24.191626245470978
Iteration 14, Best Length: 24.191626245470978
Iteration 15, Best Length: 24.191626245470978
Iteration 16, Best Length: 24.191626245470978
Iteration 17, Best Length: 24.191626245470978
Iteration 18, Best Length: 24.191626245470978
Iteration 19, Best Length: 24.191626245470978
Iteration 20, Best Length: 24.191626245470978
Iteration 21, Best Length: 24.191626245470978
Iteration 22, Best Length: 24.191626245470978
Iteration 23, Best Length: 24.191626245470978
Iteration 24, Best Length: 24.191626245470978
Iteration 25, Best Length: 24.191626245470978
Iteration 26, Best Length: 24.191626245470978
Iteration 27, Best Length: 24.191626245470978
Iteration 28, Best Length: 24.191626245470978
Iteration 29, Best Length: 24.191626245470978
Iteration 30, Best Length: 24.191626245470978
Best Tour: [4, 3, 2, 0, 1]
Best Tour Length: 24.191626245470978
```



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:

21/11/24

④ Cuckoo search Algorithm:

```
Function CuckooSearch (Func , D , N , MaxIter , pa, alpha):
    nests = RandomlyInitialize (N,D)
    fitness = EvaluateFitness (nests , Func)
    best_nest = nests [MinFitnessIndex (fitness)]
    best_fitness = Min (fitness)
    for iteration in range (1, MaxIter)
        for each nest i in nests:
            step_size = alpha * LevyFlight(i)
            new_nest = nests[i] + step_size
            new_fitness = Func (new_nest)

            if newFitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

            worst_nest = GetWorstNest (fitness, pa)
            ReplaceWorstNest (worst_nest)

        current_best_index = MinFitnessIndex (fitness)
        if fitness [current_best_index] < best_fitness:
            best_nest = nests [current_best_index]
            best_fitness = fitness [current_best_index]

    Return best_nest , best_fitness

Function LevyFlight(B):
    Returns RandomNormal (0,1,D)
```

```

function RandomlyInitialize (N,D):
    return Random(N,D)

function EvaluateFitness (nests, Func):
    return Apply(Func, nests)

function MinFitnessIndex (fitness):
    return ArgMin(fitness)

function GetWorstNests (fitness, pa):
    return SortByFitness (fitness) [int(pa*N):]

function ReplaceWorstNests (worst_nests):
    for each worst_nest in worst_nests:
        worst_nest = Random (N,D)
    
```

D - dimensionality of the problem
 N - no of nests
 pa = probability of nest abandonment
 $\alpha \Rightarrow$ scaling factor for Levy flight
 lowerbound } bounds of search space
 upperbound

Cb
 21/11/17

Code:

```

#cuckoo search
import numpy as np
import random
import math
import matplotlib.pyplot as plt

# Define a sample function to optimize (Sphere function in this case)
def objective_function(x):
  
```

```

return np.sum(x ** 2)

# Lévy flight function
def levy_flight(Lambda):
    sigma_u = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, size=1)
    v = np.random.normal(0, sigma_v, size=1)
    step = u / (abs(v) ** (1 / Lambda))
    return step

# Cuckoo Search algorithm
def cuckoo_search(num_nests=25, num_iterations=100, discovery_rate=0.25, dim=5, lower_bound=-10, upper_bound=10):
    # Initialize nests
    nests = np.random.uniform(lower_bound, upper_bound, (num_nests, dim))
    fitness = np.array([objective_function(nest) for nest in nests])

    # Get the current best nest
    best_nest_idx = np.argmin(fitness)
    best_nest = nests[best_nest_idx].copy()
    best_fitness = fitness[best_nest_idx]

    Lambda = 1.5 # Parameter for Lévy flights
    fitness_history = [] # To track fitness at each iteration

    for iteration in range(num_iterations):
        # Generate new solutions via Lévy flight
        for i in range(num_nests):
            step_size = levy_flight(Lambda)
            new_solution = nests[i] + step_size * (nests[i] - best_nest)
            new_solution = np.clip(new_solution, lower_bound, upper_bound)
            new_fitness = objective_function(new_solution)

            # Replace nest if new solution is better
            if new_fitness < fitness[i]:
                nests[i] = new_solution
                fitness[i] = new_fitness

        # Discover some nests with probability 'discovery_rate'
        random_nests = np.random.choice(num_nests, int(discovery_rate * num_nests), replace=False)
        for nest_idx in random_nests:
            nests[nest_idx] = np.random.uniform(lower_bound, upper_bound, dim)
            fitness[nest_idx] = objective_function(nests[nest_idx])

    # Update the best nest
    current_best_idx = np.argmin(fitness)
    if fitness[current_best_idx] < best_fitness:

```

```

best_fitness = fitness[current_best_idx]
best_nest = nests[current_best_idx].copy()

# Store fitness for plotting
fitness_history.append(best_fitness)

# Print the best solution at each iteration (optional)
print(f"Iteration {iteration+1}/{num_iterations}, Best Fitness: {best_fitness}")

# Plot fitness convergence graph
plt.plot(fitness_history)
plt.title('Fitness Convergence Over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Best Fitness')
plt.show()

# Return the best solution found
return best_nest, best_fitness

# Example usage
best_nest, best_fitness = cuckoo_search(num_nests=30, num_iterations=30, dim=10, lower_bound=-5, upper_bound=5)
print("Best Solution:", best_nest)
print("Best Fitness:", best_fitness)

```

OUTPUT:

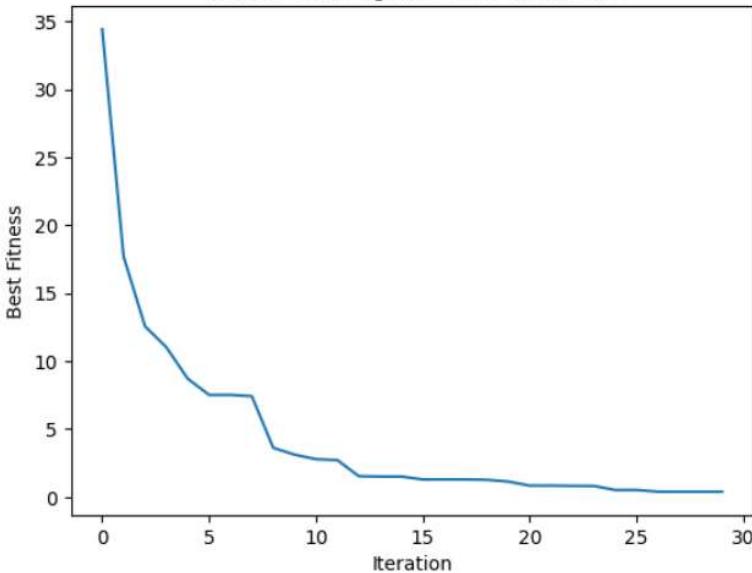
```

➡ Iteration 1/30, Best Fitness: 34.421347350368414
Iteration 2/30, Best Fitness: 17.701267864864427
Iteration 3/30, Best Fitness: 12.572246094152595
Iteration 4/30, Best Fitness: 11.025968548544025
Iteration 5/30, Best Fitness: 8.713786692960158
Iteration 6/30, Best Fitness: 7.5206125475077785
Iteration 7/30, Best Fitness: 7.5206125475077785
Iteration 8/30, Best Fitness: 7.426062303628502
Iteration 9/30, Best Fitness: 3.6305424687807872
Iteration 10/30, Best Fitness: 3.122312407680085
Iteration 11/30, Best Fitness: 2.7935374916676268
Iteration 12/30, Best Fitness: 2.7258275326189683
Iteration 13/30, Best Fitness: 1.5451154817432429
Iteration 14/30, Best Fitness: 1.5138101828809285
Iteration 15/30, Best Fitness: 1.5138101828809285
Iteration 16/30, Best Fitness: 1.300269684490209
Iteration 17/30, Best Fitness: 1.300269684490209
Iteration 18/30, Best Fitness: 1.300269684490209
Iteration 19/30, Best Fitness: 1.2738498249584989
Iteration 20/30, Best Fitness: 1.1445834652176474
Iteration 21/30, Best Fitness: 0.8487556087655604
Iteration 22/30, Best Fitness: 0.8487556087655604
Iteration 23/30, Best Fitness: 0.8289231635578032
Iteration 24/30, Best Fitness: 0.8242402471719793
Iteration 25/30, Best Fitness: 0.5258270013075049
Iteration 26/30, Best Fitness: 0.5258270013075049
Iteration 27/30, Best Fitness: 0.3996236442626478
Iteration 28/30, Best Fitness: 0.3996236442626478
Iteration 29/30, Best Fitness: 0.3996236442626478
Iteration 30/30, Best Fitness: 0.3996236442626478

```

Iteration 30/30, Best Fitness: 0.3996236442626478

Fitness Convergence Over Iterations



Best Solution: [0.1187463 -0.27226584 0.30789726 -0.04631337 -0.41395998 0.0566571
-0.05240855 -0.11176247 0.15695263 -0.00202369]
Best Fitness: 0.3996236442626478

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:

```
⑥ Grey Wolf Optimization:
Randomly initialize Grey wolf population of N particles  $x_i$ ;  

 $(i = 1, 2, \dots, n)$ 
Calculate the fitness value of each individuals
Sort grey wolf population based on fitness value;
alpha-wolf = wolf with best/least fitness value
beta-wolf = wolf with second least fitness value
gamma-wolf = wolf with third least fitness value
For Iter in range(max_iter):
    calculate the value of  $\alpha, \beta, \gamma$  using
     $\alpha = 2 + (1 - \text{Iter}/\text{maxiter})$ 
    For i in range(N):
        a. compute the values of  $A_1, A_2, A_3$  and  $C_1, C_2, C_3$ 
         $A_1 = \alpha * (2 * x_1 - 1)$ 
         $A_2 = \alpha * (2 * x_2 - 1)$ 
         $A_3 = \alpha * (2 * x_3 - 1)$ 
         $C_1 = 2 * x_1$ 
         $C_2 = 2 * x_2$ 
         $C_3 = 2 * x_3$ 
        b. compute  $x_1, x_2, x_3$ 
         $x_1 = \text{alpha\_wolf\_position} -$ 
         $A_1 * \text{abs}(C_1 * \text{alpha\_wolf\_position} - \text{ith\_wolf\_position})$ 
         $x_2 = \text{beta\_wolf\_position} -$ 
         $A_2 * \text{abs}(C_2 * \text{beta\_wolf\_position} - \text{ith\_wolf\_position})$ 
         $x_3 = \text{gamma\_wolf\_position} -$ 
```

```

AB A3 = abs(Cc3 * gamma*wolf_position - ith.wolf.position)

c. Compute new solution and its fitness
x_new = (x1 + x2 + x3) / 3
fitness_new = fitness(x_new)

d. Update the ith_wolf greedily
if (fnew < ith_wolf.fitness)
    ith_wolf.position = x_new
    ith_wolf.fitness = fnew

End-for

# compute new alpha, beta and gamma
sort_grey_wolf_population based on fitness values
alpha_wolf = wolf with least fitness value
beta_wolf = wolf with second-least fitness value
gamma_wolf = wolf with third-least fitness value

End FOR

return best wolf in the population

```

Code:

```

#GWO
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the Problem (a mathematical function to optimize)
def objective_function(x):

```

```

return np.sum(x**2) # Example: Sphere function (minimize sum of squares)

# Step 2: Initialize Parameters
num_wolves = 5 # Number of wolves in the pack
num_dimensions = 2 # Number of dimensions (for the optimization problem)
num_iterations = 30 # Number of iterations
lb = -10 # Lower bound of search space
ub = 10 # Upper bound of search space

# Step 3: Initialize Population (Generate initial positions randomly)
wolves = np.random.uniform(lb, ub, (num_wolves, num_dimensions))

# Initialize alpha, beta, delta wolves
alpha_pos = np.zeros(num_dimensions)
beta_pos = np.zeros(num_dimensions)
delta_pos = np.zeros(num_dimensions)

alpha_score = float('inf') # Best (alpha) score
beta_score = float('inf') # Second best (beta) score
delta_score = float('inf') # Third best (delta) score

# To store the alpha score over iterations for graphing
alpha_score_history = []

# Step 4: Evaluate Fitness and assign Alpha, Beta, Delta wolves
def evaluate_fitness():
    global alpha_pos, beta_pos, delta_pos, alpha_score, beta_score, delta_score

    for wolf in wolves:
        fitness = objective_function(wolf)

        # Update Alpha, Beta, Delta wolves based on fitness
        if fitness < alpha_score:
            delta_score = beta_score
            delta_pos = beta_pos.copy()

            beta_score = alpha_score
            beta_pos = alpha_pos.copy()

            alpha_score = fitness
            alpha_pos = wolf.copy()
        elif fitness < beta_score:
            delta_score = beta_score

```

```

delta_pos = beta_pos.copy()

beta_score = fitness
beta_pos = wolf.copy()
elif fitness < delta_score:
    delta_score = fitness
    delta_pos = wolf.copy()

# Step 5: Update Positions
def update_positions(iteration):
    a = 2 - iteration * (2 / num_iterations) # a decreases linearly from 2 to 0

    for i in range(num_wolves):
        for j in range(num_dimensions):
            r1 = np.random.random()
            r2 = np.random.random()

            # Position update based on alpha
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_pos[j] - wolves[i, j])
            X1 = alpha_pos[j] - A1 * D_alpha

            # Position update based on beta
            r1 = np.random.random()
            r2 = np.random.random()
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_pos[j] - wolves[i, j])
            X2 = beta_pos[j] - A2 * D_beta

            # Position update based on delta
            r1 = np.random.random()
            r2 = np.random.random()
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_pos[j] - wolves[i, j])
            X3 = delta_pos[j] - A3 * D_delta

            # Update wolf position
            wolves[i, j] = (X1 + X2 + X3) / 3

            # Apply boundary constraints

```

```

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

# Step 6: Iterate (repeat evaluation and position updating)
for iteration in range(num_iterations):
    evaluate_fitness() # Evaluate fitness of each wolf
    update_positions(iteration) # Update positions based on alpha, beta, delta

    # Record the alpha score for this iteration
    alpha_score_history.append(alpha_score)

    # Optional: Print current best score
    print(f"Iteration {iteration+1}/{num_iterations}, Alpha Score: {alpha_score}")

# Step 7: Output the Best Solution
print("Best Solution:", alpha_pos)
print("Best Solution Fitness:", alpha_score)

# Plotting the convergence graph
plt.plot(alpha_score_history)
plt.title('Convergence of Grey Wolf Optimizer')
plt.xlabel('Iteration')
plt.ylabel('Alpha Fitness Score')
plt.grid(True)
plt.show()

```

OUTPUT:

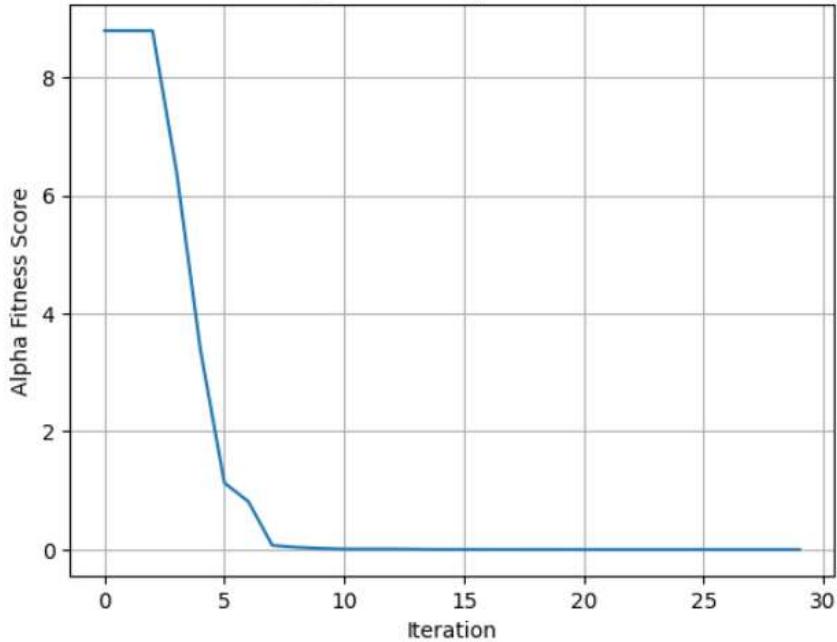
```

➡ Iteration 1/30, Alpha Score: 8.789922247101906
Iteration 2/30, Alpha Score: 8.789922247101906
Iteration 3/30, Alpha Score: 8.789922247101906
Iteration 4/30, Alpha Score: 6.409956649485766
Iteration 5/30, Alpha Score: 3.383929841190778
Iteration 6/30, Alpha Score: 1.1292299489236237
Iteration 7/30, Alpha Score: 0.8136628488047792
Iteration 8/30, Alpha Score: 0.07110881373527288
Iteration 9/30, Alpha Score: 0.03823180120070083
Iteration 10/30, Alpha Score: 0.021111314445105462
Iteration 11/30, Alpha Score: 0.00874782100259989
Iteration 12/30, Alpha Score: 0.00874782100259989
Iteration 13/30, Alpha Score: 0.00874782100259989
Iteration 14/30, Alpha Score: 0.005066807028932165
Iteration 15/30, Alpha Score: 0.0011746187200998674
Iteration 16/30, Alpha Score: 0.0011746187200998674
Iteration 17/30, Alpha Score: 0.0008078646351838173
Iteration 18/30, Alpha Score: 0.0008078646351838173
Iteration 19/30, Alpha Score: 0.0006302256737926024
Iteration 20/30, Alpha Score: 0.0005272190797352655
Iteration 21/30, Alpha Score: 0.00035614966782860404
Iteration 22/30, Alpha Score: 0.0003270119398391142
Iteration 23/30, Alpha Score: 0.00022723766847392013
Iteration 24/30, Alpha Score: 0.00022152382849585967
Iteration 25/30, Alpha Score: 0.00022152382849585967
Iteration 26/30, Alpha Score: 0.00020102313789207912
Iteration 27/30, Alpha Score: 0.0001974565833678501
Iteration 28/30, Alpha Score: 0.0001547675581999543
Iteration 29/30, Alpha Score: 0.00014751518222697009
Iteration 30/30, Alpha Score: 0.00014751518222697009
Best Solution: [ 0.00643925 -0.01029812]
Best Solution Fitness: 0.00014751518222697009

```

best solution fitness: 0.00014751518222697009

Convergence of Grey Wolf Optimizer



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:

6) Parallel cellular Algorithms & Program:

```
Function objective.function(x):
    return sum (B_i^2 for x_i in x)

Function get.neighbors(grid, i, j, grid-size):
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if (di != 0 or dj != 0):
                ni = (i + di) mod grid-size
                nj = (j + dj) mod grid-size
                neighbors.append((ni, nj))
    return neighbors

Function update.rule(current.sol, neighbor.sol, perturbation):
    for each dimension d in solution:
        new.solution[d] = neighbor.sol[d] + RANDOM(
            -perturbation.range, perturbation.range)
    return new.solution

Function Parallel.Cellular.Algorithm(grid-size, max.iter, col.dim, lb, ub):
    initialize grid with random solutions in [lb, ub]
    initialize fitness values for all cells using objective function
    set global best.sol = none
    set global best.fitness = infinity

    for t = 1 to max.iter:
        create an empty new.grid for updated sol
        for i = 0 to grid-size-1:
            for j = 0 to grid-size-1:
                neighbors = get.neighbors(grid, i, j, grid-size)
```

```

best-neighbor = grid[i][j]
best-fitness = fitness[i][j]
for each (ni, nj) in neighbors:
    if fitness[ni][nj] < best-fitness:
        best-fitness ← grid[ni][nj]
        best-neighbor ← fitness[ni][nj]

new-sol = update-rule(grid[i][j], best-neighbor,
                      perturbation-range=0.1)
new-fit = objective-function(new-sol)

if new-fit < fitness[i][j]:
    new-grid[i][j] ← new-sol
    fitness[i][j] = new-fit
else:
    new-grid[i][j] ← grid[i][j]

if fitness[i][j] < global-best-fitness:
    global-best-sol = grid[i][j]
    global-best-fitness = fitness[i][j]

grid = new-grid

return global-best-sol, global-best-fitness

```

Code:

```

#pcap
import numpy as np

# Define the problem: A simple optimization function (e.g., Sphere Function)
def optimization_function(position):
    """Example: Sphere Function for minimization."""
    return sum(x**2 for x in position)

```

```

# Initialize Parameters
GRID_SIZE = (10, 10) # Grid size (rows, columns)
NEIGHBORHOOD_RADIUS = 1 # Moore neighborhood radius
DIMENSIONS = 2 # Number of dimensions in the solution space
ITERATIONS = 30 # Number of iterations

# Initialize Population
def initialize_population(grid_size, dimensions):
    """Initialize a grid with random positions."""
    population = np.random.uniform(-10, 10, size=(grid_size[0], grid_size[1], dimensions))
    return population

# Evaluate Fitness
def evaluate_fitness(population):
    """Calculate the fitness of all cells."""
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness

# Get Neighborhood
def get_neighborhood(grid, x, y, radius):
    """Get the neighbors of a cell within the specified radius."""
    neighbors = []
    for i in range(-radius, radius + 1):
        for j in range(-radius, radius + 1):
            if i == 0 and j == 0:
                continue # Skip the current cell
            ni, nj = x + i, y + j
            if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
                neighbors.append((ni, nj))
    return neighbors

# Update States
def update_states(population, fitness):
    """Update the state of each cell based on its neighbors."""
    new_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            neighbors = get_neighborhood(population, i, j, NEIGHBORHOOD_RADIUS)
            best_neighbor = population[i, j]
            best_fitness = fitness[i, j]

            # Find the best position among neighbors
            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_fitness = fitness[ni, nj]
                    best_neighbor = population[ni, nj]
            new_population[i, j] = best_neighbor
    return new_population

```

```

# Update the cell state (move towards the best neighbor)
new_population[i, j] = (population[i, j] + best_neighbor) / 2 # Average position
return new_population

# Main Algorithm
def parallel_cellular_algorithm():
    """Implementation of the Parallel Cellular Algorithm."""
    population = initialize_population(GRID_SIZE, DIMENSIONS)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(ITERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

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

        # Update states based on neighbors
        population = update_states(population, fitness)

        # Print progress
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")

    print("\nBest Solution Found:")
    print(f"Position: {best_solution}, Fitness: {best_fitness}")

# Run the algorithm
if __name__ == "__main__":
    parallel_cellular_algorithm()

```

OUTPUT:

```
⌚ Iteration 1: Best Fitness = 0.43918427791098213
➡ Iteration 2: Best Fitness = 0.43918427791098213
Iteration 3: Best Fitness = 0.062221279350329436
Iteration 4: Best Fitness = 0.030149522005462108
Iteration 5: Best Fitness = 0.015791278460696168
Iteration 6: Best Fitness = 0.0025499667118763104
Iteration 7: Best Fitness = 0.0025499667118763104
Iteration 8: Best Fitness = 0.00019007166980743008
Iteration 9: Best Fitness = 0.00019007166980743008
Iteration 10: Best Fitness = 1.0432171933623911e-05
Iteration 11: Best Fitness = 8.406928148912647e-06
Iteration 12: Best Fitness = 5.511032710180021e-07
Iteration 13: Best Fitness = 4.3084388056725156e-07
Iteration 14: Best Fitness = 2.315054420755622e-07
Iteration 15: Best Fitness = 5.245753459404661e-08
Iteration 16: Best Fitness = 5.245753459404661e-08
Iteration 17: Best Fitness = 4.341357920017173e-08
Iteration 18: Best Fitness = 1.145644119860328e-08
Iteration 19: Best Fitness = 3.147791691706415e-09
Iteration 20: Best Fitness = 2.8192306881167533e-09
Iteration 21: Best Fitness = 9.788374665398935e-11
Iteration 22: Best Fitness = 9.788374665398935e-11
Iteration 23: Best Fitness = 9.788374665398935e-11
Iteration 24: Best Fitness = 9.788374665398935e-11
Iteration 25: Best Fitness = 7.537171686605552e-11
Iteration 26: Best Fitness = 7.234639306921671e-11
Iteration 27: Best Fitness = 7.028872029493468e-11
Iteration 28: Best Fitness = 3.340290444524624e-11
Iteration 29: Best Fitness = 1.4953679944431498e-11
Iteration 30: Best Fitness = 1.0817118995466254e-11

Best Solution Found:
Position: [-2.92599538e-06 -1.50188883e-06], Fitness: 1.0817118995466254e-11
```

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:

```
1) Optimization via Gene Expression Algorithms:  
function objective function (sol):  
    return sum (x2 for x in sol)  
  
function gene-exp (genes, seq):  
    return genes-seq #direct mapping  
  
function selection (population, fitness, selecting_size):  
    sorted_idx = sorted (range (len(fitness)), key=lambda k: fitness[k])  
    return [population[i] for i in sorted_idx[selection_idx]]  
  
function crossover (parent1, parent2, crossover_rate):  
    if random.random () < crossover_rate:  
        point<- random.randint (1, len (parent1)-1)  
        child1<- parent1[:point]+parent2[point:]  
        child2<- parent2[:point]+parent1[point:]  
    else:  
        child1, child2= parent1, parent2  
    return child1, child2  
  
function mutation (individual, mutation_rate, lb, ub):  
    for i in range (len (individual)):  
        if random.random () < mutation_rate:  
            individual[i]= random.uniform (lb, ub)  
    return individual  
  
function gene-exp-algo (population_size, gene_len, crossover_rate,  
mutation_rate, max_gen, lb, ub):  
    population=[  
        [random.uniform (lb, ub) for _ in range (gene_len)]  
        for _ in range (population_size)]  
    ]
```

```

fitness = [objective_function(gene_expr(ind)) for ind in population]
best.sol = population[0]
best.fit = fitness[0]

for gen in range(1, max_gen + 1):
    selected.popl = selection(population, fitness, population_size
                                // 2)

    offspring = []
    for i in range(0, len(selected.popl), 2):
        parent1 = selected.popl[i]
        parent2 = selected.popl[i+1] % len(selected.popl)
        child1, child2 = crossover(parent1, parent2, crossover_rate)
        offspring.append(child1)
        offspring.append(child2)

    for i in range(len(offspring)):
        offspring[i] = mutation(offspring[i], mutation_rate)

    population = selected.popl + offspring

    fitness = [objective_function(gene_expr(ind)) for ind in population]

    for i in range(len(population)):
        if fitness[i] < best_fitness:
            best.sol = population[i]
            best.fit = fitness[i]

return best.sol, best.fit

```

Code:

```

import numpy as np
import random

# 1. Define the Problem: Optimization Function (e.g., Sphere Function)
def optimization_function(solution):
    """Sphere Function for minimization (fitness evaluation)."""
    return sum(x**2 for x in solution)

# 2. Initialize Parameters

```

```

POPULATION_SIZE = 50 # Number of genetic sequences (solutions)
GENES = 5 # Number of genes per solution
MUTATION_RATE = 0.1 # Probability of mutation
CROSSOVER_RATE = 0.7 # Probability of crossover
GENERATIONS = 30 # Number of generations to evolve

# 3. Initialize Population
def initialize_population(pop_size, genes):
    """Generate initial population of random genetic sequences."""
    return np.random.uniform(-10, 10, (pop_size, genes))

# 4. Evaluate Fitness
def evaluate_fitness(population):
    """Evaluate the fitness of each genetic sequence."""
    fitness = [optimization_function(solution) for solution in population]
    return np.array(fitness)

# 5. Selection: Tournament Selection
def select_parents(population, fitness, num_parents):
    """Select parents using tournament selection."""
    parents = []
    for _ in range(num_parents):
        tournament = random.sample(range(len(population)), 3) # Randomly select 3 candidates
        best = min(tournament, key=lambda idx: fitness[idx])
        parents.append(population[best])
    return np.array(parents)

# 6. Crossover: Single-Point Crossover
def crossover(parents, crossover_rate):
    """Perform crossover between pairs of parents."""
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 >= len(parents):
            break
        parent1, parent2 = parents[i], parents[i + 1]
        if random.random() < crossover_rate:
            point = random.randint(1, len(parent1) - 1) # Single crossover point
            child1 = np.concatenate((parent1[:point], parent2[point:]))
            child2 = np.concatenate((parent2[:point], parent1[point:]))
        else:
            child1, child2 = parent1, parent2 # No crossover
        offspring.extend([child1, child2])
    return np.array(offspring)

# 7. Mutation
def mutate(offspring, mutation_rate):
    """Apply mutation to introduce variability."""
    for i in range(len(offspring)):
        for j in range(len(offspring[i])):
            if random.random() < mutation_rate:
                offspring[i][j] += random.uniform(-1, 1)

```

```

        if random.random() < mutation_rate:
            offspring[i][j] += np.random.uniform(-1, 1) # Random small change
    return offspring

# 8. Gene Expression: Functional Solution (No transformation needed for this case)
def gene_expression(population):
    """Translate genetic sequences into functional solutions."""
    return population # Genetic sequences directly represent solutions here.

# 9. Main Function: Gene Expression Algorithm
def gene_expression_algorithm():
    """Implementation of Gene Expression Algorithm for optimization."""
    # Initialize population
    population = initialize_population(POPULATION_SIZE, GENES)
    best_solution = None
    best_fitness = float('inf')

    for generation in range(GENERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

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

        # Selection
        parents = select_parents(population, fitness, POPULATION_SIZE // 2)

        # Crossover
        offspring = crossover(parents, CROSSOVER_RATE)

        # Mutation
        offspring = mutate(offspring, MUTATION_RATE)

        # Gene Expression
        population = gene_expression(offspring)

        # Print progress
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

    # Output the best solution
    print("\nBest Solution Found:")
    print(f"Position: {best_solution}, Fitness: {best_fitness}")

# 10. Run the Algorithm
if __name__ == "__main__":
    gene_expression_algorithm()

```

OUTPUT:

```
Generation 1: Best Fitness = 55.82997756903893
Generation 2: Best Fitness = 26.410565738143625
Generation 3: Best Fitness = 21.857647823851615
Generation 4: Best Fitness = 20.016914182036285
Generation 5: Best Fitness = 20.016914182036285
Generation 6: Best Fitness = 20.016914182036285
Generation 7: Best Fitness = 13.81760087982789
Generation 8: Best Fitness = 13.81760087982789
Generation 9: Best Fitness = 12.077725051361178
Generation 10: Best Fitness = 10.461698723345474
Generation 11: Best Fitness = 8.933105023570093
Generation 12: Best Fitness = 6.619449963941974
Generation 13: Best Fitness = 3.1567413435369454
Generation 14: Best Fitness = 3.1567413435369454
Generation 15: Best Fitness = 3.1567413435369454
Generation 16: Best Fitness = 2.74585545305795
Generation 17: Best Fitness = 2.7831453676198964
Generation 18: Best Fitness = 2.078188177116774
Generation 19: Best Fitness = 1.5193087227027497
Generation 20: Best Fitness = 1.4413606561895607
Generation 21: Best Fitness = 0.8501569187378994
Generation 22: Best Fitness = 0.4209372164676112
Generation 23: Best Fitness = 0.3893761873774093
Generation 24: Best Fitness = 0.3893761873774093
Generation 25: Best Fitness = 0.3893761873774093
Generation 26: Best Fitness = 0.3741053651316379
Generation 27: Best Fitness = 0.1381555631914642
Generation 28: Best Fitness = 0.12238160343023853
Generation 29: Best Fitness = 0.12238160343023853
Generation 30: Best Fitness = 0.12238160343023853

Best Solution Found:
Position: [-0.03614343 -0.00257499  0.02260677  0.31412563  0.14792784], Fitness: 0.12238160343023853
```