VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SAMARTH KUMAR DUBEY(1BM22CS235)

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 **SAMARTH KUMAR DUBEY(1BM22CS235),** 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.

Prof Sonika Sharma Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Samarth512/BIS_Lab

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.

Implementation Steps:

1. **Define the Problem**: Create a mathematical function to optimize.
2. **Initialize Parameters**: Set the population size, mutation rate, crossover rate, and number of generations.
3. **Create Initial Population**: Generate an initial population of potential solutions.
4. **Evaluate Fitness**: Evaluate the fitness of each individual in the population.
5. **Selection**: Select individuals based on their fitness to reproduce.
6. **Crossover**: Perform crossover between selected individuals to produce offspring.
7. **Mutation**: Apply mutation to the offspring to maintain genetic diversity.
8. **Iteration**: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. **Output the Best Solution**: Track and output the best solution found during the generations.

Algorithm:

Implementation-

Genetic algorithm for function $f(x) = x^2$

Pseudo code

function Genetic Algorithm

// Set parameters
Set popl^n size, mutation_rate, crossover rate, Generation, range.

// initialize population
population = Initialize Population (population size, range)

best_solution = None
best_fitness = -9999

// Loop for each generation
For generation 1 to n
// Evaluate fitness for population.
fitness = Evaluate fitness(population)

// Select best solution
max_fitness = max(Fitness)
if max_fitness > best_fitness
best_fitness = max_fitness
best_solution = population[Index of max fitness]

new population = [ ]
while (new population) < population_size

Saathi

Date __/__/____

parent1, parent2 = SelectParents(population, fitness)

child1, child2 = Crossover(parent1, parent2)

child1 = mutate(child1, range, mutation rate)

child2 = mutate(child2, range, mutation rate)

Append ~~child~~ new population, (child1, child2)

population = new population

end of loop

Print "Best solution", best fitness

End.

Function EvaluatePopulation(population)

~~for~~ ~~fitness~~

Return objective function(x) for x in population

End

Function InitializePopulation(size, range)

Return random(• in range) for ~~pop~~ size

End

Function ~~Evaluate~~ SelectParents(population, fitness)

~~return~~ total_f = Sum(fitness)

probability = f / total_f

return random(population, 2, probabilities)

End

Date __/__/____

Saathi

```
Function Crossover (parent1, parent2)
    if random < crossover
        alpha = random()
        child1 = alpha * parent1 + (1-alpha) * parent2
        child2 = alpha * parent2 + (1-alpha) * parent1
    Else
        return parent1, parent2.
End

Function mutate (individual, range, mutation)
    if random < mutation
        return random (* range)
    return individual
End
```

Code:

```
import numpy as np

def target_function(x):
    return x**2 + np.random.normal(0, 0.1)

def create_population(size, bounds):
    return np.random.uniform(bounds[0], bounds[1], size)

def calculate_fitness(population):
    return np.array([target_function(ind) for ind in population])

def select_parents(population, fitness):
    fitness_sum = np.sum(fitness)
    if fitness_sum == 0:
        return np.random.choice(population, size=2)

    probabilities = fitness / fitness_sum
    return population[np.random.choice(range(len(population)), size=2, p=probabilities)]

def crossover(parent1, parent2, crossover_rate):
    if np.random.rand() < crossover_rate:
        return (parent1 + parent2) / 2
    return parent1

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

def replacement(old_population, new_population):
    combined_population = np.concatenate((old_population, new_population))
    combined_fitness = calculate_fitness(combined_population)
    best_indices = np.argsort(combined_fitness)[-len(old_population):]
    return combined_population[best_indices]

def genetic_algorithm(pop_size, bounds, generations, mutation_rate, crossover_rate):
    population = create_population(pop_size, bounds)

    for gen in range(generations):
        fitness = calculate_fitness(population)
        best_fitness = round(np.max(fitness), 5)
        print(f"Generation {gen + 1}: Best Fitness = {best_fitness}")
```

```
        new_population = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population, fitness)
            child1 = mutate(crossover(parent1, parent2, crossover_rate), mutation_rate, bounds)
            child2 = mutate(crossover(parent2, parent1, crossover_rate), mutation_rate, bounds)
            new_population.extend([child1, child2])

        population = replacement(population, new_population)

    final_fitness = calculate_fitness(population)
    best_idx = np.argmax(final_fitness)
    best_individual = int(round(population[best_idx]))
    best_fitness = round(final_fitness[best_idx], 5)

    print(f"Best individual: {best_individual}, Fitness: {best_fitness}")

POPULATION_SIZE = 10
GENERATION_COUNT = 50
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
BOUNDS = (0, 4)

genetic_algorithm(POPULATION_SIZE, BOUNDS, GENERATION_COUNT,
MUTATION_RATE, CROSSOVER_RATE)
```

OutPut :

```
Generation 1: Best Fitness = 14.03015
Generation 2: Best Fitness = 14.25866
Generation 3: Best Fitness = 14.92493
Generation 4: Best Fitness = 16.07853
Generation 5: Best Fitness = 15.9414
Generation 6: Best Fitness = 16.0325
Generation 7: Best Fitness = 16.10904
Generation 8: Best Fitness = 16.18451
Generation 9: Best Fitness = 16.12389
Generation 10: Best Fitness = 16.11446
Generation 11: Best Fitness = 16.03909
Generation 12: Best Fitness = 16.19334
Generation 13: Best Fitness = 16.14563
Generation 14: Best Fitness = 16.13926
Generation 15: Best Fitness = 16.21125
Generation 16: Best Fitness = 16.24335
Generation 17: Best Fitness = 16.25528
Generation 18: Best Fitness = 16.17218
Generation 19: Best Fitness = 16.1357
Generation 20: Best Fitness = 16.14817
Generation 21: Best Fitness = 16.14305
Generation 22: Best Fitness = 16.0949
Generation 23: Best Fitness = 16.26894
Generation 24: Best Fitness = 16.20695
Generation 25: Best Fitness = 16.16363
Generation 26: Best Fitness = 16.21272
Generation 27: Best Fitness = 16.08122
Generation 28: Best Fitness = 16.12665
Generation 29: Best Fitness = 16.18215
Generation 30: Best Fitness = 16.10114
Generation 31: Best Fitness = 16.16454
Generation 32: Best Fitness = 16.26946
Generation 33: Best Fitness = 16.17145
Generation 34: Best Fitness = 16.12566
Generation 35: Best Fitness = 16.1072
Generation 36: Best Fitness = 16.13436
Generation 37: Best Fitness = 16.18996
Generation 38: Best Fitness = 16.22735
Generation 39: Best Fitness = 16.08516
Generation 40: Best Fitness = 16.11481
Generation 41: Best Fitness = 16.17697
Generation 42: Best Fitness = 16.1651
Generation 43: Best Fitness = 16.09014
Generation 44: Best Fitness = 16.16286
Generation 45: Best Fitness = 16.08904
Generation 46: Best Fitness = 16.18089
Generation 47: Best Fitness = 16.16205
Generation 48: Best Fitness = 16.09395
Generation 49: Best Fitness = 16.18297
Generation 50: Best Fitness = 16.07618
Best individual: 4, Fitness: 16.19651
```

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.

Implementation Steps:

1. **Define the Problem**: Create a mathematical function to optimize.
2. **Initialize Parameters**: Set the number of particles, inertia weight, cognitive and social coefficients.
3. **Initialize Particles**: Generate an initial population of particles with random positions and velocities.
4. **Evaluate Fitness**: Evaluate the fitness of each particle based on the optimization function.
5. **Update Velocities and Positions**: Update the velocity and position of each particle based on its own best position and the global best position.
6. **Iterate**: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. **Output the Best Solution**: Track and output the best solution found during the iterations.

Algorithm:

Date ___/___/_____

Algorithm : Particle swarm optimization

i) Initialize

a) Objective function, number of particles, number of dimensions, maximum iteration, bounds

b) Initialize n particle positions within bounds and velocities randomly

c) Initialize personal and global bests with starting position

ii) Iterative Process

a) update velocity

for every iteration
for every particle

Calculate $V(t) = w \cdot V(t-1) + c_1 r_1 (pbest(t-1) - x_i(t-1)) + c_2 \cdot r_2 (gbest(t-1) - x_i(t-1))$

b) update position
for every particle
$x(t) = x(t-1) + V(t)$

c) Evaluate new fitness with new position $x(t)$ for each particle.
Update personal & global best

Date __/__/____ Saathi

iii) Termination

After max iterations, algorithm returns global best position & global best value, that represents optimal solution found by swarm.

Implementation

1 Objective function

```
def obj_func(x):
    return x**2 + 2
```

2 PSO

```
def PSO(num_particle, max_iter, w, c1, c2, dim):
    # initialize particles & positions & velocities
    position = np.random.uniform(-10, 10,
        (num_particle), dim)

    #Random velocities
    velocities = np.random.uniform(-1, 1,
        (num_particle, dim))

    personal_best_position = position.copy()

    personal_best_score = np.array([objective
        _function(pos) for pos in position])

    global_best_position = personal_best_position
        [np.argmin(personal_best
```

Date __/__/____

3 Main PSO loop

for iter in range (max_iter):
    for i in range (num_particles):
        r1, r2 = np.random.rand(2)
        velocities[i] = w * velocities[i-1]
            + c1 * r1 * (personal_best
            - positions[i]) + (c2 * r2 *
            (global_best_pos - positions
        positions[i] = positions[i] + velocity[i]

        fitness = objective_func (positions[i])

        if fitness < personal_best_scores[i]:
            personal_best_scores[i] =
                fitness
            personal_best_positions[i] =
                positions[i]

4 update Global best

min_score_index = np.argmin (personal_best_sc
if (personal_best_scores[min_score_index] < global
    _best_score):
        global_best_score = personal_best_scores
            [min_score_index]
        global_best_position = personal_best_pos
            [min_score_index]
return global_best_position, global_best_score

5

5 Initialize values

num_particle = 30

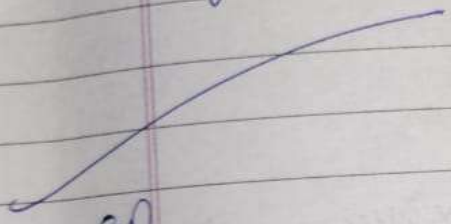
max_iter = 100

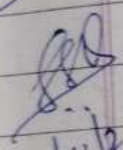
w = 0.7

c1 = 1.5

c2 = 1.5

dh = 1





Code :

```
import numpy as np

# Define the optimization function (Sphere function)
def sphere_function(x):
    return np.sum(x**2)

# PSO parameters
num_particles = 30       # Number of particles
num_dimensions = 2       # Dimensionality of the problem
num_iterations = 100     # Number of iterations
w = 0.5                  # Inertia weight
c1 = 1.5                 # Cognitive (personal) coefficient
c2 = 1.5                 # Social (global) coefficient

# Initialize particles' positions and velocities
particles_position = np.random.uniform(-10, 10, (num_particles, num_dimensions))
particles_velocity = np.random.uniform(-1, 1, (num_particles, num_dimensions))

# Initialize personal best positions and global best position
personal_best_position = particles_position.copy()
personal_best_value = np.array([sphere_function(x) for x in particles_position])
global_best_position = personal_best_position[np.argmin(personal_best_value)]
global_best_value = np.min(personal_best_value)

# PSO main loop
```

```
for t in range(num_iterations):
    for i in range(num_particles):
        # Update velocity
        r1, r2 = np.random.rand(num_dimensions), np.random.rand(num_dimensions)
        cognitive_velocity = c1 * r1 * (personal_best_position[i] - particles_position[i])
        social_velocity = c2 * r2 * (global_best_position - particles_position[i])
        particles_velocity[i] = w * particles_velocity[i] + cognitive_velocity + social_velocity

        # Update position
        particles_position[i] = particles_position[i] + particles_velocity[i]

        # Evaluate fitness
        fitness_value = sphere_function(particles_position[i])

        # Update personal best
        if fitness_value < personal_best_value[i]:
            personal_best_value[i] = fitness_value
            personal_best_position[i] = particles_position[i]

        # Update global best
        if fitness_value < global_best_value:
            global_best_value = fitness_value
            global_best_position = particles_position[i]

    # Print best value in current iteration
    print(f"Iteration {t+1}/{num_iterations}, Best Value: {global_best_value}")
```

```
# Output the best solution found
print("Best solution found:")
print("Position:",(global_best_position))
print("Value:", (global_best_value))
```

Output :

```
Iteration 46/100, Best Value: 1.8530325312352947e-11
Iteration 47/100, Best Value: 1.8530325312352947e-11
Iteration 48/100, Best Value: 1.8530325312352947e-11
Iteration 49/100, Best Value: 1.8530325312352947e-11
Iteration 50/100, Best Value: 1.8530325312352947e-11
Iteration 51/100, Best Value: 1.7400531789711l4e-11
Iteration 52/100, Best Value: 1.424714208530917e-11
Iteration 53/100, Best Value: 2.960973190313889e-14
Iteration 54/100, Best Value: 2.960973190313889e-14
Iteration 55/100, Best Value: 2.960973190313889e-14
Iteration 56/100, Best Value: 2.960973190313889e-14
Iteration 57/100, Best Value: 2.960973190313889e-14
Iteration 58/100, Best Value: 2.960973190313889e-14
Iteration 59/100, Best Value: 2.960973190313889e-14
Iteration 60/100, Best Value: 2.960973190313889e-14
Iteration 61/100, Best Value: 2.960973190313889e-14
Iteration 62/100, Best Value: 2.960973190313889e-14
Iteration 63/100, Best Value: 2.960973190313889e-14
Iteration 64/100, Best Value: 2.960973190313889e-14
Iteration 65/100, Best Value: 2.92075771196141e-14
Iteration 66/100, Best Value: 5.808300002181341e-16
Iteration 67/100, Best Value: 5.808300002181341e-16
Iteration 68/100, Best Value: 5.808300002181341e-16
Iteration 69/100, Best Value: 5.808300002181341e-16
Iteration 70/100, Best Value: 5.808300002181341e-16
Iteration 71/100, Best Value: 5.808300002181341e-16
Iteration 72/100, Best Value: 5.808300002181341e-16
Iteration 73/100, Best Value: 5.808300002181341e-16
Iteration 74/100, Best Value: 3.982905292835492e-16
Iteration 75/100, Best Value: 4.0962666995010955e-17
Iteration 76/100, Best Value: 4.0962666995010955e-17
Iteration 77/100, Best Value: 4.0962666995010955e-17
Iteration 78/100, Best Value: 4.0962666995010955e-17
Iteration 79/100, Best Value: 3.366508883506532e-17
Iteration 80/100, Best Value: 2.4601423037380456e-17
Iteration 81/100, Best Value: 2.4601423037380456e-17
Iteration 82/100, Best Value: 1.9404201911347848e-17
Iteration 83/100, Best Value: 2.9494226572905646e-18
Iteration 84/100, Best Value: 2.9494226572905646e-18
Iteration 85/100, Best Value: 2.9494226572905646e-18
Iteration 86/100, Best Value: 2.9494226572905646e-18
Iteration 87/100, Best Value: 2.9494226572905646e-18
Iteration 88/100, Best Value: 2.9494226572905646e-18
Iteration 89/100, Best Value: 2.9494226572905646e-18
Iteration 90/100, Best Value: 2.9494226572905646e-18
Iteration 91/100, Best Value: 2.9494226572905646e-18
Iteration 92/100, Best Value: 2.9494226572905646e-18
Iteration 93/100, Best Value: 6.289531829382844e-19
Iteration 94/100, Best Value: 6.289531829382844e-19
Iteration 95/100, Best Value: 6.289531829382844e-19
Iteration 96/100, Best Value: 2.687283750634462e-19
Iteration 97/100, Best Value: 2.687283750634462e-19
Iteration 98/100, Best Value: 1.78959873922167e-19
Iteration 99/100, Best Value: 1.78959873922167e-19
Iteration 100/100, Best Value: 1.1280850297663162e-19
Best solution found:
Position: [ 2.24611439e-10 -2.49716248e-10]
Value: 1.1280850297663162e-19
```

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.

Implementation Steps:

1. **Define the Problem**: Create a set of cities with their coordinates.
2. **Initialize Parameters**: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. **Construct Solutions**: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. **Update Pheromones**: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. **Iterate**: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. **Output the Best Solution**: Keep track of and output the best solution found during the iterations.

Algorithm:

Date __/__/____

Ant colony optimization

i) Initialize

~~init~~
define cities, n_ants, n_iterations,
decay, alpha, beta
~~class AntColony:~~
~~def __init__(self, cities~~

ii) Calculate distance matrix between cities,
initialize pheromone levels on all edges
between cities to a small constant,
set best path & best distance to ~~inf~~ infinite

iii) ~~Path~~ In main loop
repeat for n iterations
for each ant in n_ants
a) start at random city
b) create empty list path &
add ~~starting~~ city.
c) create a set visited to keep
track of cities visited by the
ant
d) while path is not complete
select next city from unvisited
cities using rule

$$p(i \to j) = \frac{\tau(i,j)^{\alpha} \cdot \eta(i,j)^{\beta}}{\sum_{k \in \text{unvisited}} \tau(i,k)^{\alpha} \cdot \eta(i,k)^{\beta}}$$

iv) ~~Pheromone update~~
Select a city & push it into visited

Date

After completing path, update best path, best distance

V) Pheromone update
After all paths are completed,
for each edge $(i,j) \Rightarrow \tau(i,j) \leftarrow (1-\rho) \cdot \tau(i,j)$

where ρ is decay rate

vi) for each ant
calculate amount of pheromone to deposit on edge in ant's path

$$\Delta\tau(i,j) = \frac{Q}{L_k}$$

$$\tau(i,j) \leftarrow \tau(i,j) + \Delta\tau(i,j)$$

vii) Return best path & total distance

Pseudocode

Initialize cities, pheromone matrix, distance matrix
Initialize best path to empty, best distance to infinity

for iteration = 1 to n_iterations:
  for each ant k in 1 to num_ants: {
    Randomly select a starting city
    initialize ant's path as [start_city]
    Initialize visited cities as {start_city}
  }
  while ant's path is not complete {
    calculate transition probabilities for unvisited cities
    select next city based on probability

Page No.

Saathi

Date ___/___/_____

Add next city to path
Add next city to visited cities

3

Evaporate pheromones on all edges:
for each edge (i, j):
$\tau(i,j) = (1 - decay) * \tau(i,j)$

Deposit pheromones for each ant's path:
for each ant k:
for each edge (i, j) in ant's path:
$\tau(i,j) += Q / L_k$

return best path & best_distance

SBS
14/11/25

Code :

```
import random
import math
import numpy as np

class AntColonyTSP:
    def __init__(self, cities, n_ants, n_iterations, alpha, beta, rho, tau_0):
        self.cities = cities
        self.n_cities = len(cities)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha  # Pheromone importance
        self.beta = beta  # Heuristic information importance
        self.rho = rho  # Evaporation rate
        self.tau_0 = tau_0  # Initial pheromone value
        self.dist_matrix = self.calculate_distances()  # Calculate the distance
matrix
        self.pheromone_matrix = np.full((self.n_cities, self.n_cities), tau_0)  #
Initialize pheromone matrix

    def calculate_distances(self):
        """Calculate the Euclidean distance between each pair of cities."""
        dist_matrix = np.zeros((self.n_cities, self.n_cities))
        for i in range(self.n_cities):
            for j in range(i + 1, self.n_cities):
                dist = math.sqrt((self.cities[i][0] - self.cities[j][0])**2 +
                                 (self.cities[i][1] - self.cities[j][1])**2)
                dist_matrix[i][j] = dist_matrix[j][i] = dist
        self.print_distance_matrix(dist_matrix)  # Print the distance matrix
        return dist_matrix

    def print_distance_matrix(self, dist_matrix):
        """Print the distance matrix with cities numbered starting from 1."""
        print("Distance Matrix:")
        print("        ", end="")
        for i in range(self.n_cities):
            print(f"City {i + 1}", end="   ")  # Print cities starting from 1
        print()

        for i in range(self.n_cities):
            print(f"City {i + 1}: ", end="")  # Print cities starting from 1
            for j in range(self.n_cities):
```

```
            print(f"{dist_matrix[i][j]:.2f}", end="    ")
        print()

    def select_next_city(self, ant, visited):
        """Select the next city for the ant to visit based on pheromone and
distance."""
        current_city = ant[-1]
        probabilities = []
        for next_city in range(self.n_cities):
            if next_city not in visited:
                pheromone = self.pheromone_matrix[current_city][next_city] **
self.alpha
                distance = self.dist_matrix[current_city][next_city]
                heuristic = (1.0 / distance) ** self.beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)

        total = sum(probabilities)
        probabilities = [prob / total for prob in probabilities]
        return random.choices(range(self.n_cities), probabilities)[0]

    def construct_solution(self):
        """Construct a solution for one ant."""
        visited = set()
        ant = [random.randint(0, self.n_cities - 1)]  # Start at a random city
        visited.add(ant[0])

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

        return ant

    def calculate_total_distance(self, solution):
        """Calculate the total distance of a solution."""
        total_distance = 0
        for i in range(len(solution) - 1):
            total_distance += self.dist_matrix[solution[i]][solution[i + 1]]
        total_distance += self.dist_matrix[solution[-1]][solution[0]]  # Return to
start
        return total_distance

    def update_pheromones(self, all_solutions, all_distances):
        """Update pheromones based on the solutions found by the ants."""
        # Evaporate pheromone
        self.pheromone_matrix *= (1 - self.rho)
```

```
        # Deposit pheromone
        for i in range(self.n_ants):
            solution = all_solutions[i]
            distance = all_distances[i]
            pheromone_deposit = 1.0 / distance
            for i in range(len(solution) - 1):
                self.pheromone_matrix[solution[i]][solution[i + 1]] +=
pheromone_deposit
            self.pheromone_matrix[solution[-1]][solution[0]] +=
pheromone_deposit  # Return to start

    def run(self):
        """Run the Ant Colony Optimization algorithm."""
        best_solution = None
        best_distance = float('inf')

        for _ in range(self.n_iterations):
            all_solutions = []
            all_distances = []

            # Each ant constructs a solution
            for _ in range(self.n_ants):
                solution = self.construct_solution()
                total_distance = self.calculate_total_distance(solution)
                all_solutions.append(solution)
                all_distances.append(total_distance)

                # Update best solution
                if total_distance < best_distance:
                    best_solution = solution
                    best_distance = total_distance

            # Update pheromones
            self.update_pheromones(all_solutions, all_distances)

        return best_solution, best_distance


# Example usage

# Define cities (x, y) coordinates
cities = [(0, 0), (1, 3), (4, 3), (6, 1), (5, 0), (3, 2), (2, 5), (7, 6)]

# Parameters
n_ants = 10
n_iterations = 100
alpha = 1.0  # Influence of pheromone
```

```
beta = 2.0   # Influence of distance heuristic
rho = 0.1    # Pheromone evaporation rate
tau_0 = 1.0  # Initial pheromone level


# Create the ACO solver
aco = AntColonyTSP(cities, n_ants, n_iterations, alpha, beta, rho, tau_0)

# Run the algorithm
best_solution, best_distance = aco.run()

# Output the results
print(f"\nBest solution (starting from City 1): {['City ' + str(i+1) for i in
best_solution]}")
print(f"Best distance: {best_distance:.2f}")
```

Output :

```
Distance Matrix:
        City 1   City 2   City 3   City 4   City 5   City 6   City 7   City 8
City 1: 0.00   3.16   5.00   6.08   5.00   3.61   5.39   9.22
City 2: 3.16   0.00   3.00   5.39   5.00   2.24   2.24   6.71
City 3: 5.00   3.00   0.00   2.83   3.16   1.41   2.83   4.24
City 4: 6.08   5.39   2.83   0.00   1.41   3.16   5.66   5.10
City 5: 5.00   5.00   3.16   1.41   0.00   2.83   5.83   6.32
City 6: 3.61   2.24   1.41   3.16   2.83   0.00   3.16   5.66
City 7: 5.39   2.24   2.83   5.66   5.83   3.16   0.00   5.10
City 8: 9.22   6.71   4.24   5.10   6.32   5.66   5.10   0.00

Best solution (starting from City 1): ['City 8', 'City 4', 'City 5', 'City 3', 'City 6', 'City 1', 'City 2', 'City 7']
Best distance: 25.19
```

Program 4:

Cuckoo Search (CS) Optimization

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.

Implementation Steps:

1. **Define the Problem**: Create a mathematical function to optimize.
2. **Initialize Parameters**: Set the number of nests, the probability of discovery, and the number of iterations.
3. **Initialize Population**: Generate an initial population of nests with random positions.
4. **Evaluate Fitness**: Evaluate the fitness of each nest based on the optimization function.
5. **Generate New Solutions**: Create new solutions via Lévy flights.
6. **Abandon Worst Nests**: Abandon a fraction of the worst nests and replace them with new random positions.
7. **Iterate**: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. **Output the Best Solution**: Track and output the best solution found during the iterations.

Algorithm :

Cuckoo search

Algorithm

1) Initialize

Initialize objective function $f(x)$ number of nests = n_nests, number of iteration (n_iterations), Levy flight parameter (Lambda), abandonment probability (p_a), step size scaling factor (alpha), Dimensionality of search space (dim)

2) Initialize nests randomly within the search space:

nests = random_initialization(n_nests, dim)

3) Evaluate fitness

fitness = [evaluate_fitness(nest) for each nest in nests]

4) Identify the best bee:

best_nest = nest with minimum fitness

best_fitness = minimum fitness value

5) Repeat for each iteration (1 To n_iterations):

a) for each nest i:

- Generate a new nest using Levy flight:

new_nest = nests[i] + alpha * Levy_flight(Lambda, dim)

Date ___/___/___

new_nest_i = clip(new_nest_i, lower_bound, upper_bound)

- Evaluate fitness of the new nest:
new_fitness_i = evaluate_fitness(new_nest_i)

- if the new nest is better, replace the old nest:
if new_fitness_i < fitness[i]:
nests[i] = new_nest_i
fitness[i] = new_fitness_i

6) Abandon worst nests based on probability
- For each nest, if random(0, 1) < Pa or fitness[i] is worse:
Replace nest i with new random nest

c) Update the best nest if necessary:
best_nest = nest with minimum fitness
best_fitness = minimum fitness value

6) Return best solution found:
Return best_nest.

SBD
21/11/24

Code :

```
import numpy as np
import math

# Objective Function (Optimization Problem)
# This is a sample function to optimize. Modify it as per your problem.
def objective_function(x):
    return np.sum(x**2)  # Example: Minimize the sum of squares (f(x) = sum(x^2))

# Lévy Flight Step (used to explore the solution space)
def levy_flight(Lambda, size):
    sigma_u = (math.gamma(1 + Lambda) * np.sin(math.pi * Lambda / 2) /
               math.gamma((1 + Lambda) / 2) * np.power(Lambda, 1 / 2))
    u = np.random.normal(0, sigma_u, size)
    v = np.random.normal(0, 1, size)
    step = u / np.power(np.abs(v), 1 / Lambda)
    return step

# Cuckoo Search Algorithm
def cuckoo_search(objective_function, n_nests=25, n_iterations=1000, alpha=0.01,
p_a=0.25, Lambda=1.5, dim=5):
    # Initialize nests (positions of the solutions)
    nests = np.random.uniform(-10, 10, (n_nests, dim))  # Random positions in a 10x10
space
    fitness = np.array([objective_function(nest) for nest in nests])  # Fitness of each nest

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

    for iteration in range(n_iterations):
        # Generate new solutions via Lévy flights
        new_nests = np.copy(nests)
        for i in range(n_nests):
            # Generate a new solution using Lévy flights
            step = levy_flight(Lambda, dim)
            new_nests[i] = nests[i] + alpha * step

            # Ensure the new nest is within the boundary
            new_nests[i] = np.clip(new_nests[i], -10, 10)

        # Evaluate the new solutions
```

```
        new_fitness = np.array([objective_function(nest) for nest in new_nests])

        # Abandon the worst nests and replace with new random solutions
        for i in range(n_nests):
            if new_fitness[i] < fitness[i] or np.random.rand() < p_a:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        # Update the best solution if a new better nest is found
        best_nest = nests[np.argmin(fitness)]
        best_fitness = np.min(fitness)

        # Print iteration information
        if (iteration + 1) % 100 == 0:
            print(f"Iteration {iteration+1}: Best Fitness = {best_fitness}")

    return best_nest, best_fitness

# Parameters for the algorithm
n_nests = 25  # Number of nests (population size)
n_iterations = 1000  # Number of iterations
alpha = 0.01  # Step size scaling factor
p_a = 0.25  # Probability of discovering a new nest
Lambda = 1.5  # Lévy flight exponent (controls the step distribution)
dim = 5  # Dimensionality of the problem (number of variables)

# Run the Cuckoo Search
best_nest, best_fitness = cuckoo_search(objective_function, n_nests, n_iterations, alpha,
p_a, Lambda, dim)

# Output the best solution found
print("\nBest Nest (Solution):", best_nest)
print("Best Fitness:", best_fitness)
```

Output :

```
Iteration 100: Best Fitness = 22.184333732744214
Iteration 200: Best Fitness = 19.25748572535071
Iteration 300: Best Fitness = 15.294817086770632
Iteration 400: Best Fitness = 8.00207803162981
Iteration 500: Best Fitness = 2.0182324333288624
Iteration 600: Best Fitness = 0.379165696672302
Iteration 700: Best Fitness = 0.05124632843442799
Iteration 800: Best Fitness = 0.39524011518223845
Iteration 900: Best Fitness = 0.011230471136135817
Iteration 1000: Best Fitness = 0.07234012599821558

Best Nest (Solution): [ 0.04672369 -0.21664565  0.0092467   0.13564409 -0.06882489]
Best Fitness: 0.07234012599821558
```

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.

Implementation Steps:

1. **Define the Problem**: Create a mathematical function to optimize.
2. **Initialize Parameters**: Set the number of wolves and the number of iterations.
3. **Initialize Population**: Generate an initial population of wolves with random positions.
4. **Evaluate Fitness**: Evaluate the fitness of each wolf based on the optimization function.
5. **Update Positions**: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. **Iterate**: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. **Output the Best Solution**: Track and output the best solution found during the iterations.

Algorithm :

Date

Grey wolf optimization

Algorithm

1) Initialize parameter population size, dimensionality, lower & upper bound of solver, & maximum iteration

2) Initialize all position of wolves randomly and identify some positions as alpha, beta and gamma based on fitness function

3) Update position of other wolves based on ~~each~~ alpha, beta & delta wolves.

using two factors

a) Distance calculation: The distance b/w current position of guiding wolves alpha, Beta, Delta.

b) Exploration and Exploitation: ~~A~~ controlled by two factors, A (exploration coefficient) & C (exploitation coefficient)

D => Dimension, X => position

$$D_\alpha = |C_1 \cdot X_\alpha(t) - X_1(t)|$$

$$X_1(t+1) = X_\alpha(t) - A_1 D_\alpha$$

X_α => position of α wolf

~~A~~ similarly position for ~~the~~ beta & Beta wolves are updated

Page No.

Saathi

Date __/__/____

4) After updates, position are clipped to stay on bound.

5) After maximum iteration are reached, algorithm stops.

Application => ~~GWO can be used~~

→ Biomedical research : GWO is used in modelling modular granular neural networks

→ Wireless Sensor networks => GWO is used to address problems for wireless sensor network.

Code :

```
import numpy as np

class GreyWolfOptimizer:
    def __init__(self, obj_function, dim, lb, ub, population_size=30,
max_iter=100):
        self.obj_function = obj_function
        self.dim = dim
        self.lb = lb
        self.ub = ub
        self.population_size = population_size
        self.max_iter = max_iter

        # Initialize the positions of the wolves
        self.positions = np.random.uniform(lb, ub, (population_size, dim))
        self.alpha_pos = np.zeros(dim)
        self.beta_pos = np.zeros(dim)
        self.delta_pos = np.zeros(dim)
        self.alpha_score = float('inf')
        self.beta_score = float('inf')
        self.delta_score = float('inf')

    def optimize(self):
        for iteration in range(self.max_iter):
            for i in range(self.population_size):
                # Calculate the fitness of each wolf
                fitness = self.obj_function(self.positions[i])

                # Update Alpha, Beta, and Delta wolves
                if fitness < self.alpha_score:
                    self.alpha_score = fitness
                    self.alpha_pos = self.positions[i].copy()
                elif fitness < self.beta_score:
                    self.beta_score = fitness
                    self.beta_pos = self.positions[i].copy()
                elif fitness < self.delta_score:
                    self.delta_score = fitness
                    self.delta_pos = self.positions[i].copy()

            # Update the positions of wolves
            for i in range(self.population_size):
                for d in range(self.dim):
                    # Generate random numbers
                    r1, r2 = np.random.rand(), np.random.rand()
                    A1 = 2 * r1 - 1  # coefficient for exploration/exploitation
                    C1 = 2 * r2      # coefficient for attraction to alpha wolf
```

```
                r1, r2 = np.random.rand(), np.random.rand()
                A2 = 2 * r1 - 1
                C2 = 2 * r2

                r1, r2 = np.random.rand(), np.random.rand()
                A3 = 2 * r1 - 1
                C3 = 2 * r2

                # Calculate the distances from the alpha, beta, and delta wolves
                D_alpha = abs(C1 * self.alpha_pos[d] - self.positions[i, d])
                D_beta = abs(C2 * self.beta_pos[d] - self.positions[i, d])
                D_delta = abs(C3 * self.delta_pos[d] - self.positions[i, d])

                # Calculate the new positions for each wolf
                X1 = self.alpha_pos[d] - A1 * D_alpha
                X2 = self.beta_pos[d] - A2 * D_beta
                X3 = self.delta_pos[d] - A3 * D_delta

                # Update the position by averaging the attraction from all three
wolves
                self.positions[i, d] = (X1 + X2 + X3) / 3

            # Clamp positions to stay within the bounds
            self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)

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

    return self.alpha_pos, self.alpha_score

# Example Usage
def sphere_function(x):
   return np.sum(x**2)

gwo = GreyWolfOptimizer(obj_function=sphere_function, dim=5, lb=-10, ub=10,
population_size=30, max_iter=50)
best_position, best_score = gwo.optimize()
print("Best Position:", best_position)
print("Best Score:", best_score)
```

Output :

```
Generation 1: Best Fitness = 14.34775
Generation 2: Best Fitness = 14.32671
Generation 3: Best Fitness = 14.49234
Generation 4: Best Fitness = 14.49828
Generation 5: Best Fitness = 14.55672
Generation 6: Best Fitness = 14.55423
Generation 7: Best Fitness = 14.48036
Generation 8: Best Fitness = 14.53179
Generation 9: Best Fitness = 16.13716
Generation 10: Best Fitness = 15.93086
Generation 11: Best Fitness = 16.09927
Generation 12: Best Fitness = 16.35004
Generation 13: Best Fitness = 16.06671
Generation 14: Best Fitness = 16.10586
Generation 15: Best Fitness = 16.03114
Generation 16: Best Fitness = 16.14713
Generation 17: Best Fitness = 16.1142
Generation 18: Best Fitness = 16.07266
Generation 19: Best Fitness = 16.0683
Generation 20: Best Fitness = 16.04721
Generation 21: Best Fitness = 16.14037
Generation 22: Best Fitness = 15.99784
Generation 23: Best Fitness = 16.11707
Generation 24: Best Fitness = 16.00427
Generation 25: Best Fitness = 16.09823
Generation 26: Best Fitness = 16.09977
Generation 27: Best Fitness = 16.18316
Generation 28: Best Fitness = 16.19172
Generation 29: Best Fitness = 16.14199
Generation 30: Best Fitness = 16.08512
Generation 31: Best Fitness = 16.12503
Generation 32: Best Fitness = 16.0563
Generation 33: Best Fitness = 16.14116
Generation 34: Best Fitness = 16.10875
Generation 35: Best Fitness = 16.07223
Generation 36: Best Fitness = 16.16
Generation 37: Best Fitness = 16.13314
Generation 38: Best Fitness = 16.12542
Generation 39: Best Fitness = 16.18574
Generation 40: Best Fitness = 16.14181
Generation 41: Best Fitness = 16.12434
Generation 42: Best Fitness = 16.05498
Generation 43: Best Fitness = 16.05837
Generation 44: Best Fitness = 16.11661
Generation 45: Best Fitness = 16.07196
Generation 46: Best Fitness = 16.15789
Generation 47: Best Fitness = 16.17652
Generation 48: Best Fitness = 16.12622
Generation 49: Best Fitness = 16.20949
Generation 50: Best Fitness = 16.17873
Best individual: 4, Fitness: 16.1344
```

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.

Implementation Steps:

1. **Define the Problem**: Create a mathematical function to optimize.
2. **Initialize Parameters**: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. **Initialize Population**: Generate an initial population of cells with random positions in the solution space.
4. **Evaluate Fitness**: Evaluate the fitness of each cell based on the optimization function.
5. **Update States**: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. **Iterate**: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. **Output the Best Solution**: Track and output the best solution found during the iterations.

Algorithm:

Date __/__/____

Saathi

Parallel Cellular organism Algorithm

Algorithm steps

i) Initialize a grid of cells with random solution

ii) Define evaluation function to evaluate fitness

iii) ~~Each cell~~ Initialize a fitness array that'll store fitness calculated evaluation function for each cell

iv) For interaction with parallel cells, a cell may copy a neighbour's state (mimicry), swap state (competition), combine state (collaboration)

v) After each interact cells update their state depending on problem constraints

vi) For each generation, cells with higher solutions (higher fitness) are selected to propagate their state to neighbouring cells

vii) The process continues until iterations end

Saathi

Date __/__/____

Pseudo code

```
initialize_grid()
evaluate_fitness()

while not termination_condition():
    for each cell in grid:
        neighbor = get_neighbor(cell)
        new_state = update_cell(
        cell, neighbor)
        cell.state = new_state
    Evaluate_fitness()
    select_fittest_cells()
    update_population()
    generation += 1
```

Applications

⇒ Wireless Networks
→ Sensor placement: Optimize the placement of sensors in a grid to maximize signal coverage or data collection efficiency

⇒ Environmental & urban planning: Resource allocation optimize resource allocation

Code:

```
import numpy as np
import random

# Define any optimization function to minimize (can be changed as needed)
def custom_function(x):
    # Example function: x^2 to minimize
    return np.sum(x ** 2)  # Ensuring the function works for multidimensional
inputs

# Initialize population of genetic sequences (each individual is a sequence of
genes)
def initialize_population(population_size, num_genes, lower_bound,
upper_bound):
    # Create a population of random genetic sequences
    population = np.random.uniform(lower_bound, upper_bound, (population_size,
num_genes))
    return population

# Evaluate the fitness of each individual (genetic sequence) in the population
def evaluate_fitness(population, fitness_function):
    fitness = np.zeros(population.shape[0])
    for i in range(population.shape[0]):
        fitness[i] = fitness_function(population[i])  # Apply the fitness function to
each individual
    return fitness

# Perform selection: Choose individuals based on their fitness (roulette wheel
selection)
def selection(population, fitness, num_selected):
    # Select individuals based on their fitness (higher fitness, more likely to be
selected)
    probabilities = fitness / fitness.sum()  # Normalize fitness to create selection
probabilities
    selected_indices = np.random.choice(range(len(population)),
size=num_selected, p=probabilities)
    selected_population = population[selected_indices]
    return selected_population

# Perform crossover: Combine pairs of individuals to create offspring
def crossover(selected_population, crossover_rate):
    new_population = []
    num_individuals = len(selected_population)
    for i in range(0, num_individuals - 1, 2):  # Iterate in steps of 2, skipping the
```

```
last one if odd
        parent1, parent2 = selected_population[i], selected_population[i + 1]
        if len(parent1) > 1 and random.random() < crossover_rate:  # Only perform
crossover if more than 1 gene
            crossover_point = random.randint(1, len(parent1) - 1)  # Choose a random
crossover point
            offspring1 = np.concatenate((parent1[:crossover_point],
parent2[crossover_point:]))
            offspring2 = np.concatenate((parent2[:crossover_point],
parent1[crossover_point:]))
            new_population.extend([offspring1, offspring2])  # Create two offspring
        else:
            new_population.extend([parent1, parent2])  # No crossover, retain the
parents

    # If the number of individuals is odd, carry the last individual without crossover
    if num_individuals % 2 == 1:
        new_population.append(selected_population[-1])
    return np.array(new_population)

# Perform mutation: Introduce random changes in offspring
def mutation(population, mutation_rate, lower_bound, upper_bound):
    for i in range(population.shape[0]):
        if random.random() < mutation_rate:  # Apply mutation based on the rate
            gene_to_mutate = random.randint(0, population.shape[1] - 1)  # Choose a
random gene to mutate
            population[i, gene_to_mutate] = np.random.uniform(lower_bound,
upper_bound)  # Mutate the gene
    return population

# Gene expression: In this context, it is how we decode the genetic sequence into
a solution
def gene_expression(individual, fitness_function):
    return fitness_function(individual)

# Main function to run the Gene Expression Algorithm
def gene_expression_algorithm(population_size, num_genes, lower_bound,
upper_bound,
                              max_generations, mutation_rate, crossover_rate,
fitness_function):
    # Step 2: Initialize the population of genetic sequences
    population = initialize_population(population_size, num_genes, lower_bound,
upper_bound)
    best_solution = None
    best_fitness = float('inf')

    # Step 9: Iterate for the specified number of generations
    for generation in range(max_generations):
```

```
        # Step 4: Evaluate fitness of the current population
        fitness = evaluate_fitness(population, fitness_function)

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

        # Step 5: Perform selection (choose individuals based on fitness)
        selected_population = selection(population, fitness, population_size // 2)  #
Select half of the population

        # Step 6: Perform crossover to generate new individuals
        offspring_population = crossover(selected_population, crossover_rate)

        # Step 7: Perform mutation on the offspring population
        population = mutation(offspring_population, mutation_rate, lower_bound,
upper_bound)

        # Print output every 10 generations
        if (generation + 1) % 10 == 0:
            print(f"Generation {generation + 1}/{max_generations}, Best Fitness:
{best_fitness}")

    # Step 10: Output the best solution found
    return best_solution, best_fitness

# Parameters for the algorithm
population_size = 50  # Number of individuals in the population
num_genes = 1  # Number of genes (for a 1D problem, this is just 1, extendable
for higher dimensions)
lower_bound = -5  # Lower bound for the solution space
upper_bound = 5  # Upper bound for the solution space
max_generations = 100  # Number of generations to evolve the population
mutation_rate = 0.1  # Mutation rate (probability of mutation per gene)
crossover_rate = 0.7  # Crossover rate (probability of crossover between two
parents)

# Run the Gene Expression Algorithm
best_solution, best_fitness = gene_expression_algorithm(
    population_size, num_genes, lower_bound, upper_bound,
    max_generations, mutation_rate, crossover_rate, custom_function)

# Output the best solution found
print("\nBest Solution Found:", best_solution)
print("Best Fitness Value:", best_fitness)
```

Output :

```
Generation 10/100, Best Fitness: 0.0002226428362338621
Generation 20/100, Best Fitness: 0.0002226428362338621
Generation 30/100, Best Fitness: 0.0002226428362338621
Generation 40/100, Best Fitness: 0.0002226428362338621
Generation 50/100, Best Fitness: 0.0002226428362338621
Generation 60/100, Best Fitness: 0.0002226428362338621
Generation 70/100, Best Fitness: 0.0002226428362338621
Generation 80/100, Best Fitness: 0.0002226428362338621
Generation 90/100, Best Fitness: 0.0002226428362338621
Generation 100/100, Best Fitness: 0.0002226428362338621

Best Solution Found: [0.01492122]
Best Fitness Value: 0.0002226428362338621
```

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.

Implementation Steps:

1. **Define the Problem**: Create a mathematical function to optimize.
2. **Initialize Parameters**: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. **Initialize Population**: Generate an initial population of random genetic sequences.
4. **Evaluate Fitness**: Evaluate the fitness of each genetic sequence based on the optimization function.
5. **Selection**: Select genetic sequences based on their fitness for reproduction.
6. **Crossover**: Perform crossover between selected sequences to produce offspring.
7. **Mutation**: Apply mutation to the offspring to introduce variability.
8. **Gene Expression**: Translate genetic sequences into functional solutions.
9. **Iterate**: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. **Output the Best Solution**: Track and output the best solution found during the iterations.

Algorithm:

Saathi

Date __/__/____

Gene Expression Algorithm

1. Define the fitness function $f(x)$ =>
   I/P : Genetic sequence (x)
   O/P : fitness value (real number)

2. Initialize parameters
   population size = N
   num_Genes = G
   lower_bound, upper_bound = L, U
   max_generations = M
   mutation rate = R_mut
   crossover rate = R_cross

3. Initialize Tracking variables :
   best_solution = None
   best_fitness = Infinity

4. Initialize population :
   population = Randomly Generate N individuals with G genes in bounds [L, U]

5. For generation in range (1, M+1)
   a) Evaluate fitness :
      fitness = calculate fitness for each individual in population using $f(x)$

   b) Track the best solution :
      If min(fitness) < best_fitness :
      best_fitness = min(fitness)
      best_solution = Individual with min

Page No. ____

Date __/__/____

c) Perform selection:
Selected Population = Select top N/2 Indl. based on fitness

d) Perform Crossover:
offspring - population = Generate offspring using crossover on selected population. Combine genes of pairs of parents based on R_Cross

e) Perform mutation:
Apply random mutation to offspring population based on R_mut

f) update population:
Population = offspring - population

g) output best solution by returning best solution & best_fitness

Application

=> portfolio optimization: Allocating resources across a set of investments to maximize return minimize risk

=> Route optimization: Solving travelling salesman problems, Metrical routing etc.

18/12/24

Code:

```
import numpy as np
import random

# Step 1: Define the Problem (Optimization Function)
def fitness_function(position):
    """Example fitness function: Sphere function"""
    return sum(x**2 for x in position)

# Step 2: Initialize Parameters
grid_size = (10, 10)  # Grid size (10x10 cells)
dim = 2  # Dimensionality of each cell's position
minx, maxx = -10.0, 10.0  # Search space bounds
max_iterations = 50  # Number of iterations

# Step 3: Initialize Population (Random positions)
def initialize_population(grid_size, dim, minx, maxx):
    population = np.zeros((grid_size[0], grid_size[1], dim))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            population[i, j] = [random.uniform(minx, maxx) for _ in range(dim)]
    return population

# Step 4: Evaluate Fitness (Calculate fitness for each cell)
def evaluate_fitness(population):
    fitness_grid = np.zeros((grid_size[0], grid_size[1]))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            fitness_grid[i, j] = fitness_function(population[i, j])
    return fitness_grid

# Step 5: Update States (Update each cell based on its neighbors)
def get_neighbors(i, j):
    """Returns the coordinates of neighboring cells."""
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if not (di == 0 and dj == 0):  # Exclude the cell itself
                ni, nj = (i + di) % grid_size[0], (j + dj) % grid_size[1]
                neighbors.append((ni, nj))
    return neighbors

def update_cell(population, fitness_grid, i, j, minx, maxx):
    """Update the state of a cell based on the average state of its neighbors."""
    neighbors = get_neighbors(i, j)
    best_neighbor = min(neighbors, key=lambda x: fitness_grid[x[0], x[1]])
```

```
    # Update cell position to move towards the best neighbor's position
    new_position = population[best_neighbor[0], best_neighbor[1]] + \
                   np.random.uniform(-0.1, 0.1, dim)  # Small random perturbation

    # Ensure the new position stays within bounds
    new_position = np.clip(new_position, minx, maxx)
    return new_position

# Step 6: Iterate (Repeat for a fixed number of iterations)
population = initialize_population(grid_size, dim, minx, maxx)
for iteration in range(max_iterations):
    fitness_grid = evaluate_fitness(population)

    # Update each cell in parallel (simultaneously)
    new_population = np.zeros_like(population)
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            new_population[i, j] = update_cell(population, fitness_grid, i, j, minx,
maxx)

    population = new_population

    # Print best fitness at each iteration
    best_fitness = np.min(fitness_grid)
    print(f"Iteration {iteration + 1}, Best Fitness: {best_fitness}")

# Step 7: Output the Best Solution
best_index = np.unravel_index(np.argmin(fitness_grid), fitness_grid.shape)
best_position = population[best_index[0], best_index[1]]
best_fitness = np.min(fitness_grid)
print("Best Position Found:", best_position)
print("Best Fitness Found:", best_fitness)
```

Output :

```
Iteration 1, Best Fitness: 2.0826803502492166
Iteration 2, Best Fitness: 1.7744398596087352
Iteration 3, Best Fitness: 1.4883681724573383
Iteration 4, Best Fitness: 1.387002652076188
Iteration 5, Best Fitness: 1.2427437690805143
Iteration 6, Best Fitness: 1.0599616727655872
Iteration 7, Best Fitness: 0.8679138915700586
Iteration 8, Best Fitness: 0.7270295605596662
Iteration 9, Best Fitness: 0.5602986423626427
Iteration 10, Best Fitness: 0.4288173108004688
Iteration 11, Best Fitness: 0.29608524463484226
Iteration 12, Best Fitness: 0.24338248635399468
Iteration 13, Best Fitness: 0.1653629256411268
Iteration 14, Best Fitness: 0.09981682496633203
Iteration 15, Best Fitness: 0.03965954346480205
Iteration 16, Best Fitness: 0.007780710681381793
Iteration 17, Best Fitness: 0.00029849766888554087
Iteration 18, Best Fitness: 0.0004899086297133481
Iteration 19, Best Fitness: 4.251173519935761e-05
Iteration 20, Best Fitness: 6.0338848084739565e-05
Iteration 21, Best Fitness: 3.6146911539659715e-05
Iteration 22, Best Fitness: 9.196981175899878e-05
Iteration 23, Best Fitness: 1.938276328999423e-05
Iteration 24, Best Fitness: 0.00011841546744371104
Iteration 25, Best Fitness: 7.587836680690964e-05
Iteration 26, Best Fitness: 9.106396162522501e-05
Iteration 27, Best Fitness: 0.00048545538760096476
Iteration 28, Best Fitness: 2.8078803253508404e-05
Iteration 29, Best Fitness: 7.200201265540344e-05
Iteration 30, Best Fitness: 0.0001614676021904634
Iteration 31, Best Fitness: 0.0003952025944379032
Iteration 32, Best Fitness: 3.239940240533401e-05
Iteration 33, Best Fitness: 0.00019472366858021127
Iteration 34, Best Fitness: 9.766847442440533e-05
Iteration 35, Best Fitness: 2.8878862975346795e-05
Iteration 36, Best Fitness: 2.0488363026836332e-05
Iteration 37, Best Fitness: 5.6206471847444956e-05
Iteration 38, Best Fitness: 0.00010026031741971453
Iteration 39, Best Fitness: 2.351888244594138e-05
Iteration 40, Best Fitness: 6.908851328290999e-05
Iteration 41, Best Fitness: 9.091741874917746e-05
Iteration 42, Best Fitness: 0.00012684003938990727
Iteration 43, Best Fitness: 1.5983340123524183e-05
Iteration 44, Best Fitness: 0.0004872436461115473 5
Iteration 45, Best Fitness: 0.0002983477901214220 3
Iteration 46, Best Fitness: 1.0771428049026778e-05
Iteration 47, Best Fitness: 4.5911759331353676e-05
Iteration 48, Best Fitness: 4.446408549143865e-05
Iteration 49, Best Fitness: 0.00016271325878503776
Iteration 50, Best Fitness: 5.0893201170283464e-05
Best Position Found: [0.02679162 0.00329233]
Best Fitness Found: 5.0893201170283464e-05
```