VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems

(23CS5BSBIS) *Submitted by*

Rani Aishwarya H S (1BM22CS217)

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 **Rani Aishwarya H S (1BM22CS217),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

Sowmya T Assistant Professor Department of CSE, BMSCE	Dr.Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

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

1.Genetic Algorithm

A **genetic algorithm (GA)** is a search heuristic inspired by the process of natural selection and genetics. It is used to solve optimization and search problems. The algorithm simulates the process of natural evolution, where the fittest individuals are selected to reproduce and pass their genes to the next generation, leading to the gradual improvement of solutions.

Algorithm:

Lab-02

program - 1. 04/10/24

python implementation of a genetic algorithm to solve a mathematical problem.

```
import numpy as np
import random

fitness = lambda x: x * np.sin(x)

population_size, generations = 10,20
crossover_rate, mutation_rate = 0.8,0.1
x_min, x_max = 0,10

population = np.random.uniform(x_min,x_max,
    population_size)

for generation in range(generations):
    fitness_values = np.array([fitness(x) for
x in population])
    fitness_values = fitness_values - np.min(
fitness_values)
    total_fitness = np.sum(fitness_values)
    selection_probs = fitness_values / total_
fitness if total_fitness > 0 else np.ones(
population_size) / population_size
    selected_population = np.random.
choice(population, size=population_size,
```

Page

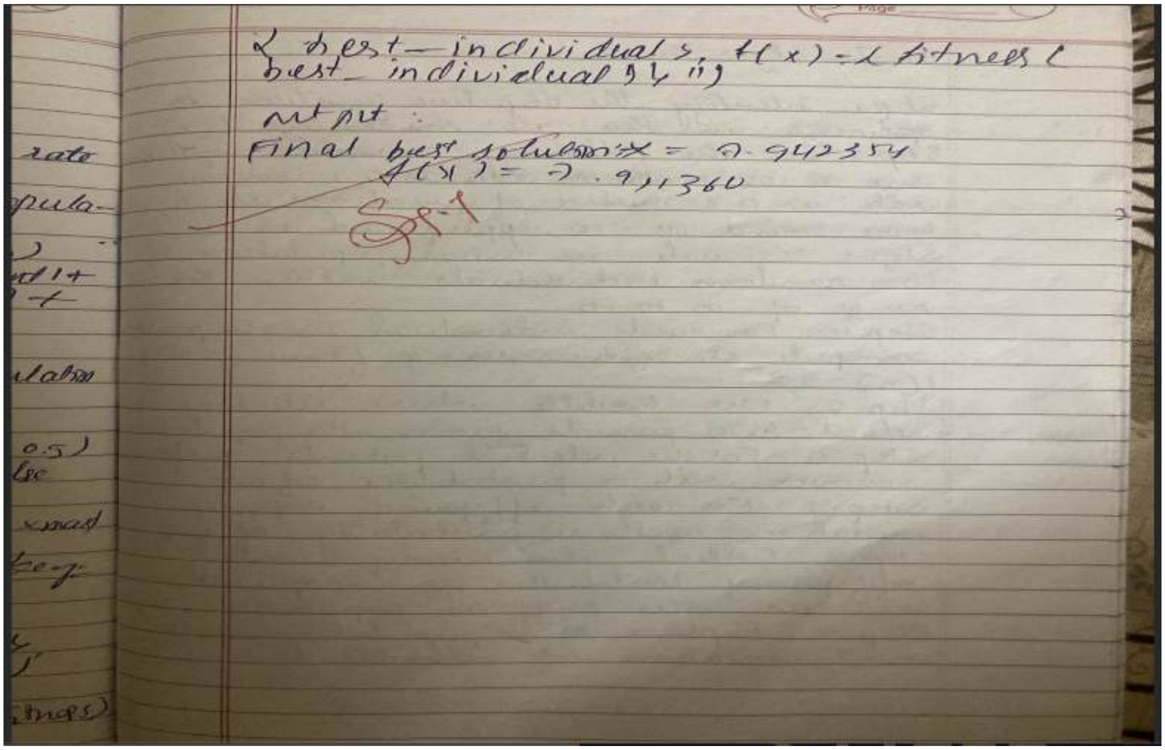
p= selection_probs)

offspring = []
for i in range (0, population_size, 2):
    if random.random() < crossover_rate and i+1 < population_size:
        parent1, parent2 = selected_population [i], selected_population [i+1]
        alpha = random.uniform (0,1)
        offspring.extend([alpha* parent1+ (1-alpha)* parent2, alpha* parent2+ (1-alpha)* parent1])
    else:
        offspring.extend([selected_population [i], selected_population [i+1]])

offspring = [x + random.uniform(-0.5, 0.5) if random.random()< mutation_rate else x for x in offspring]
population = np.clip (offspring, x_min, x_max)

best_individual = max(population, key=fitness)
print (f"generation {generation+1}: Best Solution x = {best_individual}, f(x) = {fitness(best_individual)}")

Best_individual = max(population, key=fitness)
print (f"\nFinal Best Solution: x=



{ best_individuals, f(x) = { fitness(best_individual)} ")

output :

Final best solution: x = 7.942354

f(x) = 7.941360

Code:

```
import random

# Define the fitness function
def fitness_function(x):
    return x ** 2

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

# Selection - select individuals based on fitness
def selection(population, fitness_values):
    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    selected = random.choices(population, weights=probabilities, k=len(population))
    return selected

# Crossover - create new offspring by combining parents
def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
```

```
        alpha = random.random()
        child1 = alpha * parent1 + (1 - alpha) * parent2
        child2 = alpha * parent2 + (1 - alpha) * parent1
        return child1, child2
    else:
        return parent1, parent2

# Mutation - introduce random variations
def mutate(individual, mutation_rate, lower_bound, upper_bound):
    if random.random() < mutation_rate:
        individual += random.uniform(-1, 1)
        individual = max(lower_bound, min(upper_bound, individual))  # Keep within bounds
    return individual

# Genetic Algorithm
def genetic_algorithm(population_size, lower_bound, upper_bound, generations, mutation_rate,
crossover_rate):
    population = generate_population(population_size, lower_bound, upper_bound)

    for generation in range(generations):
        # Evaluate fitness
        fitness_values = [fitness_function(ind) for ind in population]

        # Selection
        selected_population = selection(population, fitness_values)

        # Crossover
        next_generation = []
        for i in range(0, len(selected_population), 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1 if i + 1 < len(selected_population) else 0]
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            next_generation.extend([child1, child2])

        # Mutation
        population = [mutate(ind, mutation_rate, lower_bound, upper_bound) for ind in next_generation]

        # Log best fitness of the generation
        best_fitness = max(fitness_values)
        # print(f"Generation {generation + 1}: Best Fitness = {best_fitness:.4f}")

    # Return the best fitness value from the final generation
    return max(fitness_function(ind) for ind in population)
```

```
# Parameters
population_size = 10
lower_bound = -10
upper_bound = 10
generations = 50
mutation_rate = 0.1
crossover_rate = 0.8
print("Rani Aishwarya H S,1BM22CS217")
# Run Genetic Algorithm
best_fitness = genetic_algorithm(population_size, lower_bound, upper_bound, generations, mutation_rate, crossover_rate)
print(f"Best fitness found: {best_fitness:.4f}")
```

Output:

```
Rani Aishwarya H S,1BM22CS217
Best fitness found: 54.4101

=== Code Execution Successful ===
```

2.Particle Swarm Optimisation for function Optimisation

Particle Swarm Optimization (PSO) is a heuristic optimization algorithm inspired by the social behavior of birds flocking or fish schooling. It is used to find optimal solutions by mimicking the collective behavior of a swarm of particles in a search space.

Algorithm:

2 particle swarm optimization

Algorithm:

step 1: Pick a mathematical function $f(x)$ to optimize.

step 2: Set the parameters N, w, c_1, c_2 where c_1 is weight of global best position of personal best position and c_2 is the weight of global best position

step 3: Define the limits within which particle can move

step 4: Assign N with random velocity

step 5: For each particle calculate its fitness that is the best position

step 6: Update velocity based on the best velocity of its own based on the best velocity found by the swarm

step 7: It undergoes iterations to check the best solution found

step 8: Then in the final iteration it finds out the best value

Code:

```
import numpy as np
# Define the objective function
def f(x):
    return x**2

n_particles = 30
```

Code:

```
import random

# Objective function to minimize (Example: Sphere function)
def objective_function(x):
    return sum(x_i ** 2 for x_i in x)

# Particle classto represent each particle
class Particle:
    def init (self, dimension, bounds):
        self.position = [random.uniform(bounds[0], bounds[1]) for _ in
        range(dimension)]  self.velocity = [random.uniform(-1, 1) for _ in
        range(dimension)] self.pBest = list(self.position)
        self.pBest_fitness = objective_function(self.position)

# PSO class
class PSO:
    def init (self, dimension, bounds, num_particles=30, max_iterations=100):
        self.dimension = dimension
        self.bounds = bounds
        self.num_particles = num_particles
        self.max_iterations = max_iterations
        self.particles = [Particle(dimension, bounds) for _ in
        range(num_particles)]  self.gBest = list(self.particles[0].position)
        self.gBest_fitness = self.particles[0].pBest_fitness
        self.w = 0.5 # Inertia weight
        self.c1 = 1.5 # Cognitive coefficient
        self.c2 = 1.5 # Social coefficient

    def optimize(self):
        for iteration in range(self.max_iterations):
            for particle in self.particles:
                fitness = objective_function(particle.position)

                # Update personal best (pBest)
                if fitness < particle.pBest_fitness:
                    particle.pBest = list(particle.position)
                    particle.pBest_fitness = fitness
```

```
        # Update global best (gBest)
        if fitness < self.gBest_fitness:
            self.gBest = list(particle.position)
            self.gBest_fitness = fitness

    # Update velocity and position for each particle


for particle in self.particles:
for i in range(self.dimension):
# Update velocity
r1, r2 = random.random(), random.random()
particle.velocity[i] = (self.w * particle.velocity[i]
+ self.c1 * r1 * (particle.pBest[i] - particle.position[i])
+ self.c2 * r2 * (self.gBest[i] - particle.position[i]))
# Update position
particle.position[i] += particle.velocity[i]

# Ensure position stays within bounds
particle.position[i] = max(self.bounds[0], min(particle.position[i], self.bounds[1]))  return

self.gBest, self.gBest_fitness

# Define parameters
dimension = 2 # Number of dimensions
bounds = (-10, 10) # Search space bounds for each dimension
num_particles = 30 # Number of particles in the swarm
max_iterations = 100 # Maximum number of iterations
print("Rani Aishwarya H S,1BM22CS217")

# Create PSO instance and optimize
pso = PSO(dimension, bounds, num_particles, max_iterations)
best_position, best_fitness = pso.optimize()

# Output the result
print(f"Best Position: {best_position}")
print(f"Best Fitness: {best_fitness}")
```

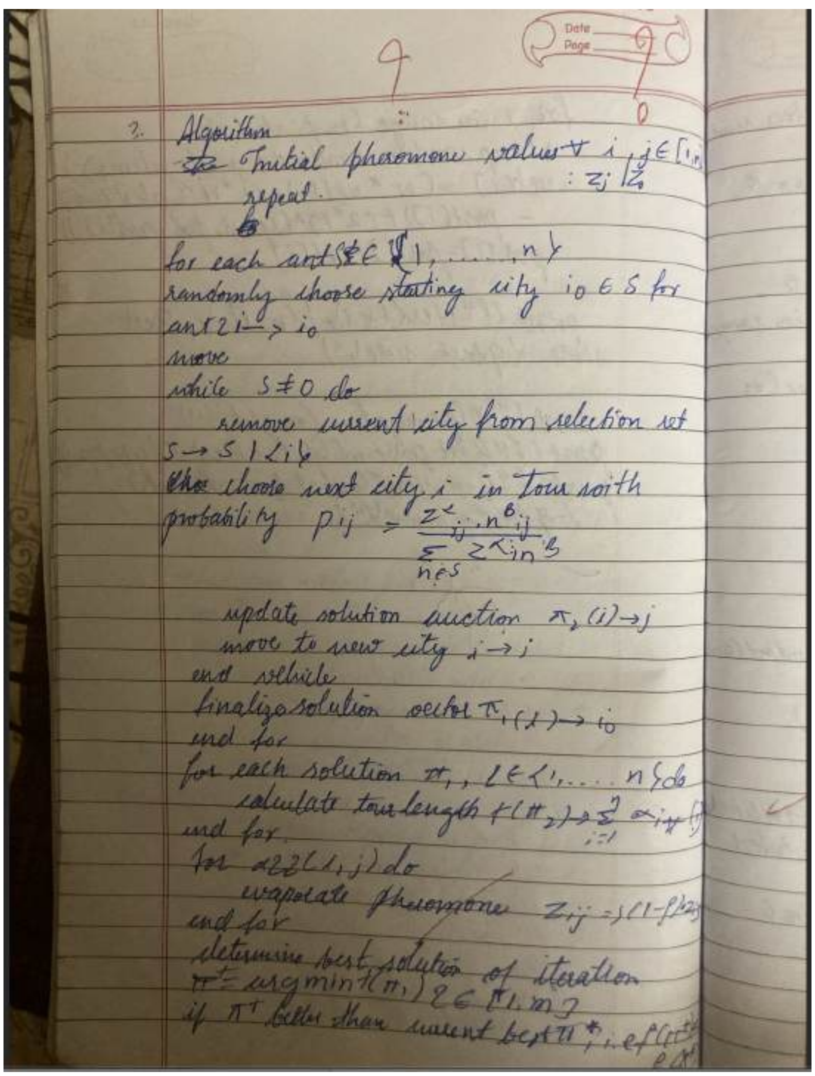
Output:

```
Rani Aishwarya H S,1BM22CS217
Best Position: [5.039947602443779e-13, 8.939424256650232e-13]
Best Fitness: 1.0531437787576535e-24
```

3.Ant Colony Optimisation

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

Algorithm:


Algorithm
Initial pheromone values ∀ i, j ∈ [1,n] : z_{ij} ← z_0
repeat
for each ant S ∈ {1, ..., n}
randomly choose starting city i_0 ∈ S for ant
i → i_0
move
while S ≠ 0 do
remove current city from selection set
S → S \ {i}
choose next city j in Tour with probability $p_{ij} = \frac{z^{\alpha}_{ij} \cdot n^{\beta}_{ij}}{\sum_{n \in S} z^{\alpha}_{in} n^{\beta}}$
update solution vector $\pi_l(i) \to j$
move to new city i → j
end while
finalize solution vector $\pi_l(l) \to i_0$
end for
for each solution π_l, l ∈ {1, ... n} do
calculate tour length $f(\pi_l) \to \sum_{i=1}^{n} \alpha_{i\pi}$
end for
for all (i, j) do
evaporate pheromone $z_{ij} \to (1-\rho) z_{ij}$
end for
determine best solution of iteration
$\pi^{t} = \arg\min f(\pi_l)$, l ∈ [1, m]
if π^{t} better than current best π^{*}, i.e. $f(\pi^{t})$ ...

then set $\pi^* \mapsto \pi^+$
end if
for all $(i,j) \in \pi^+$ do
    reinforce $z_{ij} \mapsto z_{ij} + \Delta/2$
end for
for all $(i,j) \in \pi^*$ do
    reinforce $z_{ij} \to z_{ij} + \Delta/2$
end for
until condition for termination.

program:

```
import numpy as np.

def initialize_pheromonents(n, tau0):
    return np.full((n,n), tau0)

def calculate_distance(city1, city2)
    return np.linalg.norm(city1-city2)

def calculate_probabilities(pheromones,
distances, alpha, beta, visited, current_city)
    n = len(pheromones)
    probabilities = np.zeros(n)
    for j in range(n):
        if j not in visited:
            probabilities[j] = (pheromones[
current_city], j]**alpha) * ((1/distances
[current_city, j])**beta)
```

Code:

```
import numpy as np

# Parameters
NUM_CITIES = 10 # Number of cities
NUM_ANTS = 20 # Number of ants
ITERATIONS = 10 # Number of iterations
ALPHA = 1.0 # Pheromone importance
BETA = 2.0 # Heuristic importance
EVAPORATION_RATE = 0.5
Q = 100 # Pheromone deposit factor
```

```
# Distance matrix
distance_matrix = np.random.randint(1, 100,size=(NUM_CITIES, NUM_CITIES))
np.fill_diagonal(distance_matrix, 0)

# Initialize pheromone levels
pheromones = np.ones((NUM_CITIES, NUM_CITIES))

def calculate_route_length(route):
    length = 0
    for i in range(len(route) - 1):
        length += distance_matrix[route[i], route[i + 1]]
    length += distance_matrix[route[-1], route[0]] # Return to the start city
    return length

def construct_route(start_city):
    route = [start_city]
    for _ in range(NUM_CITIES - 1):
        current_city = route[-1]
        probabilities = []
        for next_city in range(NUM_CITIES):
            if next_city not in route:
                prob = (pheromones[current_city, next_city] ** ALPHA) * \
                       ((1 / distance_matrix[current_city, next_city]) ** BETA)
                probabilities.append(prob)
            else:
                probabilities.append(0)
        probabilities = np.array(probabilities)
        probabilities /= probabilities.sum()
        next_city = np.random.choice(range(NUM_CITIES),
        p=probabilities)  route.append(next_city)
    return route

def update_pheromones(pheromones, all_routes, all_lengths):


    pheromones *= (1 - EVAPORATION_RATE) # Evaporation
    for route, length in zip(all_routes, all_lengths):
        pheromone_deposit = Q / length
        for i in range(len(route) - 1):
            pheromones[route[i], route[i + 1]] += pheromone_deposit
            pheromones[route[i + 1], route[i]] += pheromone_deposit
        # Closing the route (return to start city)
        pheromones[route[-1], route[0]] += pheromone_deposit
        pheromones[route[0], route[-1]] += pheromone_deposit
```

```
def aco():
    best_route = None
    best_length = float('inf')

    for _ in range(ITERATIONS):
        all_routes = []
        all_lengths = []

        for _ in range(NUM_ANTS):
            start_city = np.random.randint(0, NUM_CITIES)
            route = construct_route(start_city)
            route_length = calculate_route_length(route)

            all_routes.append(route)
            all_lengths.append(route_length)

            if route_length < best_length:
                best_length = route_length
                best_route = route

        update_pheromones(pheromones, all_routes, all_lengths)

    return best_route, best_length

# Run the ACO algorithm
print("Rani Aishwarya H S,1BM22CS217")
best_route, best_length = aco()
print("Best Route:", best_route)
print("Best Length:", best_length)
```

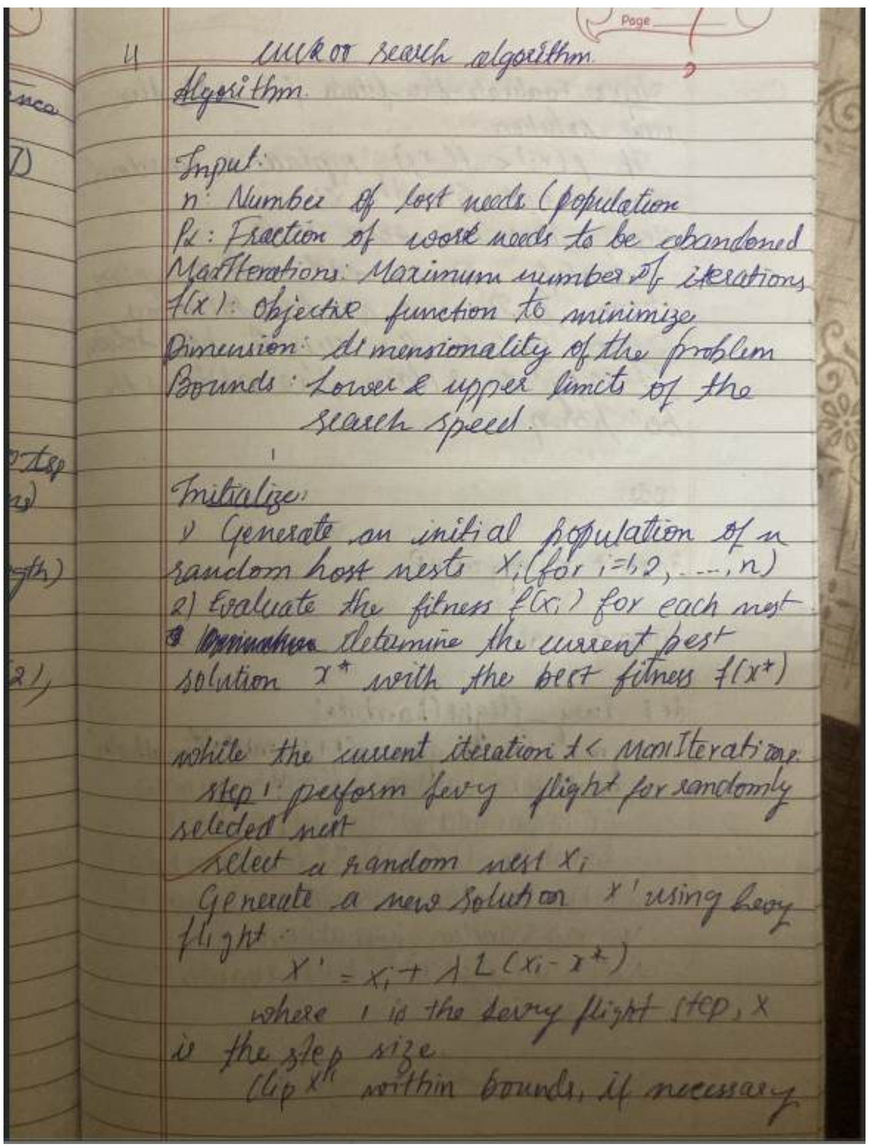
Output:

```
Rani Aishwarya H S,1BM22CS217
Best Route: [7, np.int64(1), np.int64(2), np.int64(0), np.int64(6), np
    .int64(9), np.int64(8), np.int64(4), np.int64(5), np.int64(3)]
Best Length: 220
```

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:



4 cuckoo search algorithm

Algorithm

Input:
n: Number of host nests (population
Pa: Fraction of worst nests to be abandoned
MaxIterations: Maximum number of iterations
f(x): Objective function to minimize
Dimension: dimensionality of the problem
Bounds: Lower & upper limits of the search space.

Initialize:
1) Generate an initial population of n random host nests x_i (for $i=1,2,\ldots,n$)
2) Evaluate the fitness $f(x_i)$ for each nest.
3) ~~Minimalize~~ determine the current best solution x^* with the best fitness $f(x^*)$

while the current iteration $t <$ MaxIterations:
step 1: perform levy flight for randomly selected nest
Select a random nest x_i
Generate a new solution x' using levy flight:
$$x' = x_i + \lambda L(x_i - x^*)$$
where L is the levy flight step, λ is the step size.
Clip x' within bounds, if necessary

Step4 Evaluate the fitness $f(x')$ of the new solution.
If $f(x') < f(x_j)$, replace a randomly chosen nest x_j with x'.
Step5: Abandon worse nests.
Identify $p_a \cdot n$ worst nests & replace them with new random solutions.
Step6: Update the current best solution
Identify & retain the nest with best fitness

code:

```
import numpy as np

import math

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

Code:

```
import numpy as np
import math

# Objective function (example: Sphere function, you can replace
it)  def objective_function(x):
   return sum(x**2) # Minimize the sum of squares

def levy_flight(beta, d):
    sigma_u = (math.gamma(1 + beta) * math.sin(math.pi * beta / 2) /
```

```
    (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2)))**(1 / beta) u =
    np.random.normal(0, sigma_u, d) # Draw from Gaussian distribution  v =
    np.random.normal(0, 1, d)
    step = u / (abs(v)**(1 / beta))
    return step

# Cuckoo Search Algorithm
def cuckoo_search(n, d, alpha, pa, maxGen):
    # n: Population size, d: Dimension of the problem
    # alpha: Step size, pa: Discovery probability, maxGen: Max iterations


    nests = np.random.uniform(-10, 10, (n, d))
    fitness = np.array([objective_function(nest) for nest in nests])


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

    beta = 1.5

    # Step 2: Iterative loop
    for gen in range(maxGen):
        for i in range(n):
            # Generate a new solution via Lévy flight
            step = levy_flight(beta, d)
            new_nest = nests[i] + alpha * step * (nests[i] - best_nest)
            new_nest = np.clip(new_nest, -10, 10) # Keep solutions within bounds

            # Evaluate new fitness
            new_fitness = objective_function(new_nest)
            if new_fitness < fitness[i]: # Replace with better solution
                nests[i] = new_nest
                fitness[i] = new_fitness



     # Abandon some nests with a probability pa
     for i in range(n):
     if np.random.rand() < pa:
     # Replace with new random solution
     nests[i] = np.random.uniform(-10, 10, d)
     fitness[i] = objective_function(nests[i])
```

```
# Update the current best
best_nest_index = np.argmin(fitness)
if fitness[best_nest_index] < best_fitness:
best_nest = nests[best_nest_index]
best_fitness = fitness[best_nest_index]

# print(f"Generation {gen+1}, Best Fitness: {best_fitness:.5f}")


return best_nest, best_fitness

n = 25
d = 5
alpha = 0.01
pa = 0.25
maxGen = 100

print("Rani Aishwarya H S,1BM22CS217")
best_solution, best_value = cuckoo_search(n, d, alpha, pa, maxGen)
print("Best Solution:", best_solution)
print("Best Fitness Value:", best_value)
```

Output:

```
Rani Aishwarya H S,1BM22CS217
Best Solution: [-3.2589848  -8.11177023 -1.11252639 -8.80045712 -5
    .55698494]
Best Fitness Value: 16.76397643399841
```

5.Grey Wolf Optimiser:

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

Algorithm:

Lab 5: Grey wolf optimiser:

Algorithm:

Initialize the population of wolves (positions) randomly within the search space
Define the maximum no. of iterations (T) and population size (N)
Define the fitness function to evaluate solutions.

Evaluate the fitness of each wolf in the population identify the alpha (best solution), beta (second best) and delta (third best) wolves

For t=1 to T:
    for each wolf i in the population
        for each dimension d
            A1 = 2 * a * rand() - a
            C1 = 2 * rand()
            ~~D_alpha = C1 * x_alpha[d] - x_i[d]~~
            xi = x_alpha[d] - A1 * D_alpha

            A2 = 2 * a * rand() - a
            C2 = 2 * rand()
            D_beta = C2 * x_beta[d] - x_i[d]
            X2 = x_beta[d] - A2 * D_beta

```
A3 = 2*a* rand() - a
C3 = 2* rand()
D_delta = C3 * X_delta[d] - X_i[d]
X3 = X_delta[d] - A3 * D_delta

X_i[d] = (X1+X2+X3)/3
End For
End For
a=2-(2* t/T)
Update alpha, beta & delta values
based on fitness
End For
```

Code:

import numpy as np

```
def objective_function(x):
    return x ** 2 # The function to minimize

def initialize_wolves(num_wolves, search_space):
    return np.random.uniform(search_space[0], search_space[1], num_wolves)

def update_position(alpha, beta, delta, wolf, a):
    r1, r2 = np.random.rand(), np.random.rand()
    A = 2 * a * r1 - a
    C = 2 * r2
    D = abs(C * alpha - wolf)
    X1 = alpha - A * D

    r1, r2 = np.random.rand(), np.random.rand()
    A = 2 * a * r1 - a
    C = 2 * r2
    D = abs(C * beta - wolf)
    X2 = beta - A * D

    r1, r2 = np.random.rand(), np.random.rand()
    A = 2 * a * r1 - a
    C = 2 * r2
    D = abs(C * delta - wolf)
    X3 = delta - A * D

    return (X1 + X2 + X3) / 3

def grey_wolf_optimization(obj_func, num_wolves=5, max_iter=50, search_space=(-10,
    10)):  # Initialize wolves' positions
    wolves = initialize_wolves(num_wolves, search_space)
    fitness = np.array([obj_func(wolf) for wolf in wolves])

    # Identify alpha, beta, delta
    sorted_indices = np.argsort(fitness)
    alpha, beta, delta = wolves[sorted_indices[0]], wolves[sorted_indices[1]],

    wolves[sorted_indices[2]]  a = 2 # Initial value for the parameter a

    for iteration in range(max_iter):
        for i in range(num_wolves):
            wolves[i] = update_position(alpha, beta, delta, wolves[i], a)
             wolves[i] = np.clip(wolves[i], search_space[0], search_space[1]) # Ensure wolves stay within
bounds
```

```
# Recalculate fitness and update alpha, beta, delta
fitness = np.array([obj_func(wolf) for wolf in wolves])
sorted_indices = np.argsort(fitness)
alpha, beta, delta = wolves[sorted_indices[0]], wolves[sorted_indices[1]],
wolves[sorted_indices[2]]

# Decrease a linearly
a = 2 - (2 * (iteration / max_iter))

# print(f"Iteration {iteration+1}: Alpha = {alpha}, Fitness = {obj_func(alpha)}")

return alpha, obj_func(alpha)

# Run the algorithm
print("Rani Aishwarya H S,1BM22CS217")
best_position, best_fitness = grey_wolf_optimization(objective_function)
print(f"Best Position: {best_position}")
print(f"Best Fitness: {best_fitness}")
```

Output:

```
Rani Aishwarya H S,1BM22CS217
Best Position: -5.175888459194711e-06
Best Fitness: 2.6789821342025004e-11
```

6.Parallel Cellular Algorithms and Programs:

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

Algorithm:

lab 6: parallel cellular algorithms & programs

Algorithm

1. Initialization
i) Define the grid dimensions
ii) initialize each cell based on the problem
iii) Define the neighborhood for each cell
iv) Define transition rule

2. Set initial condition
. populate the grid with all initial state for all cells

3. parallel update
- For each time step:
For each C(i,j,...) in grid (in parallel):
1. Receive the current state of C(i,j,...)
2. Retrieve the states of the neighboring cells
3. Apply the transition rule to compute the next state
if C(i,j,...)
update all cells simultaneously to their new states.

4. Check stopping condition
- Terminate if a predefined condition is met:
· A fixed no. of iterations completed
· System reaches a steady state

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):
    """Returnsthe 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 towardsthe 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("Rani Aishwarya H S,1BM22CS217")
print("Best Position Found:", best_position)
print("Best Fitness Found:", best_fitness)
```

Output:

```
Rani Aishwarya H S,1BM22CS217
Best Position Found: [ 0.03240588 -0.03531405]
Best Fitness Found: 4.3671675326705975e-06
```

7.Gene Expression Algorithms(GEA):

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:

Lab 7 optimisation via gene Expression Algorithm

1) Initialization
Define constraints & parameters:
Set population size P, no. of generation g, mutation rate M, crossover rate C and maximum tree depth D
• Define the function set F (e.g. +, -, *, /) and terminal set T.

Initialize the population:
Generate P random individual, each represented by a mathematical expression of depth up to D

2. Evaluate Fitness
1) For each individual in the population
Replace the variable x in the individuals expression with a specific value (e.g. x = 3)
2) Evaluate the mathematical expression to calculate fitness
3) If the expression is invalid, assign a high fitness value.

3) Selection
1) Identify the best individual in the population.

Code:

```
import random
import operator
import math

# Constants for the genetic algorithm
POPULATION_SIZE = 100
GENERATIONS = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
MAX_TREE_DEPTH = 5
FUNCTIONS = ['+', '*', '/']
TERMINALS = ['x', '1', '2', '3']

# Class to represent an individual in the population
class Individual:
    def __init__(self, expression):
        self.expression = expression
        self.fitness = float('inf')

    # Function to evaluate the fitness of an individual
    def evaluate_fitness(self, x_value):
        try:
            expr = self.expression.replace('x', str(x_value))
            # Using eval to evaluate the expression
            self.fitness = eval(expr)
        except Exception as e:
            self.fitness = float('inf')

# Function to generate a random individual
def generate_random_individual():
    expression = generate_random_expression(MAX_TREE_DEPTH)
    return Individual(expression)

# Function to generate a random expression (tree-like structure)
def generate_random_expression(depth):
    if depth == 0 or random.random() < 0.3:
        # Return a terminal (e.g., x or constants)
        return random.choice(TERMINALS)
    else:
        # Return a function with two subexpressions
        function = random.choice(FUNCTIONS)
        left = generate_random_expression(depth - 1)
```

```
        right = generate_random_expression(depth - 1)
        return f"({left} {function} {right})"

# Function to perform crossover between two individuals
def crossover(parent1, parent2):
    # For simplicity, we just swap subexpressions between two individuals
    expr1, expr2 = parent1.expression, parent2.expression
    split1 = random.choice(expr1.split())
    split2 = random.choice(expr2.split())
    offspring_expr = expr1.replace(split1, split2, 1)
    return Individual(offspring_expr)

# Function to mutate an individual
def mutate(individual):
    if random.random() < MUTATION_RATE:
        # Replace a random part of the expression with a new one
        mutated_expr = individual.expression
        split_expr = mutated_expr.split()
        mutated_expr = mutated_expr.replace(random.choice(split_expr),
generate_random_expression(MAX_TREE_DEPTH), 1)
        individual.expression = mutated_expr

# Function to select the best individual
def select_best_individual(population, x_value):
    best_individual = min(population, key=lambda ind: ind.fitness)
    best_individual.evaluate_fitness(x_value)
    return best_individual

# Main function to run the GEP algorithm
def run_gep_algorithm():
    population = [generate_random_individual() for _ in range(POPULATION_SIZE)]

    for generation in range(GENERATIONS):
        # Evaluate fitness for each individual
        for individual in population:
            individual.evaluate_fitness(3)  # Example with x=3

        # Select the best individual
        best_individual = select_best_individual(population, 3)

        # Print the fitness of the best individual in each generation
        print(f"Generation {generation + 1}: Best fitness = {best_individual.fitness}")

        # Create a new population using crossover and mutation
        new_population = []
```

```
    while len(new_population) < POPULATION_SIZE:
        if random.random() < CROSSOVER_RATE:
            parent1 = random.choice(population)
            parent2 = random.choice(population)
            offspring = crossover(parent1, parent2)
            new_population.append(offspring)
        else:
            individual = random.choice(population)
            mutate(individual)
            new_population.append(individual)

    population = new_population

# Run the algorithm
if __name__ == "__main__":
    print("Rani Aishwarya H S,1BM22CS217")
    run_gep_algorithm()
```

Output:

```
Rani Aishwarya H S,1BM22CS217
Generation 1: Best fitness = 0.012698412698412698
Generation 2: Best fitness = 0.017543859649122806
<string>:1: SyntaxWarning: 'float' object is not callable; perhaps you
    missed a comma?
<string>:1: SyntaxWarning: 'float' object is not callable; perhaps you
    missed a comma?
<string>:1: SyntaxWarning: 'float' object is not callable; perhaps you
    missed a comma?
Generation 3: Best fitness = 0.038461538461538464
Generation 4: Best fitness = 0.5
<string>:1: SyntaxWarning: 'float' object is not callable; perhaps you
    missed a comma?
<string>:1: SyntaxWarning: 'int' object is not callable; perhaps you
    missed a comma?
Generation 5: Best fitness = 13.5

=== Code Execution Successful ===
```