VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

CHANDRAKALA K M (1BM23CS403)

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 **CHANDRAKALA K M (1BM23CS403),** 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.

Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/Chandrakala8050/BIS_LAB.git

Program 1

Genetic Algorithm for Optimization Problems.

Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as optimizing mathematical function.

Algorithm:

LAB-1

A Genetic Algorithm is a Search Heuristic inspired by the process of natural selection. It is used to find approximate solutions to optimization & search problems. The basic idea is to evolve a population of candidate solutions over generation using operators such as selection, crossover & Mutation.

Key components of genetic Algorithm

1. Population: A set of candidate solutions.
2. Chromosome: A representation of a candidate solution.
3. Fitness Function: A function to evaluate how good a solution is.
4. Selection: The process of choosing individuals from population to create offspring.
5. Crossover: combining parts of two parent to create new offspring.
6. Mutation - Randomly altering a solution to maintain genetic diversity.
7. Termination Condition: A condition that ends the algorithm. (eg: a set number of generations or a satisfactory fitness level)

4. Sheduling problems
Application → Job Sheduling in Manufacturing.
Optimization method → GAs can optimize the sequence of jobs on machines to minimize makespan or total completion time.

5. Portfolio Optimization
Application → Investment portfolio selection.
Optimization method → GAs can maximize returns while minimizing risk by selecting optimal asset combinations based on historical data.

8/10

O/P seen

Code :

```
import numpy as np
import random

# Define the problem: The function to optimize
def fitness_function(x):     return x * np.sin(x)

#    Generate    the    initial    population    def
create_population(size, x_min, x_max):      return
np.random.uniform(x_min,     x_max,     size)     #
```

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

Selection: Roulette wheel selection def select_parents(population, fitness): fitness = fitness - np.min(fitness) + 1e-6 # Shift fitness values to be positive total_fitness = np.sum(fitness)

probabilities = fitness / total_fitness # Normalize to sum to 1

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

Crossover: Single-point crossover def crossover(parent1, parent2, crossover_rate): if random.random() < crossover_rate: point = random.randint(0, 1) # Single-point crossover for simplicity return (parent1, parent2) if point == 0 else (parent2, parent1) return parent1, parent2

Mutation: Apply random changes def mutate(individual, mutation_rate, x_min, x_max): if random.random() < mutation_rate: mutation_value = np.random.uniform(-1, 1) individual += mutation_value

individual = np.clip(individual, x_min, x_max) # Ensure within bounds return individual

Main Genetic Algorithm def genetic_algorithm(population_size, mutation_rate, crossover_rate, num_generations, x_min, x_max):

population = create_population(population_size, x_min, x_max) best_solution = None

best_fitness = -np.inf

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

Track the best solution max_fitness_index = np.argmax(fitness) if fitness[max_fitness_index] > best_fitness: best_fitness = fitness[max_fitness_index] best_solution = population[max_fitness_index]

new_population = [] for _ in range(population_size // 2): # Produce new population parent1, parent2 = select_parents(population, fitness)
offspring1, offspring2 = crossover(parent1, parent2, crossover_rate)
offspring1 = mutate(offspring1, mutation_rate, x_min, x_max) offspring2 = mutate(offspring2, mutation_rate, x_min, x_max) new_population.extend([offspring1, offspring2])

population = np.array(new_population)

return best_solution, best_fitness

Take Genetic Algorithm parameters as inputs population_size = int(input("Enter the population size: ")) mutation_rate = float(input("Enter the mutation rate (0 to 1): ")) crossover_rate = float(input("Enter the crossover rate (0 to 1): ")) num_generations = int(input("Enter the number of generations: ")) x_min = float(input("Enter the minimum value of x: ")) x_max = float(input("Enter the maximum value of x: "))

Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm(population_size, mutation_rate, crossover_rate, num_generations, x_min, x_max) print(f"Best Solution: x = {best_solution}") print(f"Best Fitness: f(x) = {best_fitness}")

Output:

```
Enter the population size: 10
Enter the mutation rate (0 to 1): 0.10
Enter the crossover rate (0 to 1): 0.8
Enter the number of generations: 50
Enter the minimum value of x: 0
Enter the maximum value of x: 10
Best Solution: x = 8.208916912223948
Best Fitness: f(x) = 7.697246776822652
```

Program 2
Particle Swarm Optimization for Function Optimization.

Implement the PSO algorithm using Python to Travelling Salesman Problem.

Algorithm :

LAB-3

24/10/24

Algorithm 2 : Particle Swarm optimization for Function optimization.

i) Define the objective function:
$f(x)$ is the mathematical function to optimize.

ii) Initialize Parameters.

a) set number of parameters N, inertia weight w, cognitive coefficient C_1, social coefficient C_2 and maximum iterations MaxIter.

iii) Initialize Particles.

a) Randomly initialize each particle position as X_i and velocity V_i.

b) set each particle's personal best $P_i = X_i$ and evaluate fitness.

c) Track global best G, the best position that is found so far.

iv) Iterate (for each iteration):

a) update velocity:

$$V_i = w \cdot V_i + C_1 \cdot rand() \cdot (P_i - X_i) + C_2 \cdot rand() \cdot (G - X_i)$$

b) update position:

$$X_i = X_i + V_i$$

c) Evaluate fitness:
update P_i and G if better fitness is found.

v) Repeat : Continue until maximum iterations or convergence.

vi) Output : the global best solution G. Return the output.

24/10

TSP

O/P

Code :

```
import numpy as np

# Function to calculate the total distance of a route (path) def
calculate_total_distance(route, distance_matrix):    total_distance = 0    for
i in range(len(route) - 1):        total_distance += distance_matrix[route[i],
route[i + 1]]     total_distance += distance_matrix[route[-1], route[0]]  #
Return to start     return total_distance

# Particle Swarm Optimization (PSO) for TSP class
PSO_TSP:
    def __init__(self, distance_matrix, num_particles=30, num_iterations=100, w=0.5, c1=1, c2=1):
        self.num_particles  =  num_particles
self.num_iterations  =  num_iterations
self.distance_matrix  =  distance_matrix
self.num_cities  =  len(distance_matrix)
self.w = w  # Inertia weight       self.c1 = c1
# Cognitive coefficient        self.c2 = c2  #
Social coefficient
```

Initialize particles' positions (routes) and velocities

self.particles = np.array([np.random.permutation(self.num_cities) for _ in range(num_particles)])

self.velocities = np.array([np.zeros(self.num_cities) for _ in range(num_particles)])

Evaluate fitness of each particle (route)

self.fitness = np.array([calculate_total_distance(route, distance_matrix) for route in self.particles])

Initialize personal best positions and fitness self.p_best = np.copy(self.particles)

self.p_best_fitness = np.copy(self.fitness)

Initialize global best position and fitness

self.g_best = self.p_best[np.argmin(self.p_best_fitness)] self.g_best_fitness = np.min(self.p_best_fitness)

Update velocities and positions def update_particles(self): for i in range(self.num_particles):

Update velocity: w * velocity + c1 * random() * (personal best - current position) + c2 * random() * (global best - current position) r1 = np.random.rand(self.num_cities) r2 = np.random.rand(self.num_cities) cognitive_velocity = self.c1 * r1 * (self.p_best[i] - self.particles[i]) social_velocity = self.c2 * r2 * (self.g_best - self.particles[i]) inertia_velocity = self.w * self.velocities[i]

self.velocities[i] = inertia_velocity + cognitive_velocity + social_velocity

To ensure we move to a new route, modify the velocity to shuffle positions velocity_order = np.argsort(self.velocities[i]) # Sort based on the velocity magnitude new_particle = np.array([self.particles[i][j] for j in velocity_order])

Ensure the new particle is a valid permutation

self.particles[i] = new_particle self.fitness[i] = calculate_total_distance(new_particle, self.distance_matrix)

Update personal best if self.fitness[i] < self.p_best_fitness[i]: self.p_best[i] = self.particles[i] self.p_best_fitness[i] = self.fitness[i]

Update global best if self.fitness[i] < self.g_best_fitness: self.g_best = self.particles[i] self.g_best_fitness = self.fitness[i]

Run the PSO algorithm

def run(self):

for iteration in range(self.num_iterations): self.update_particles()

print(f"Iteration {iteration + 1}: Best Distance = {self.g_best_fitness}")

return self.g_best, self.g_best_fitness

Function to take user input for distance matrix and PSO parameters def input_pso_parameters():

Input the number of cities and distance matrix num_cities = int(input("Enter the number of cities: ")) print("Enter the distance matrix row by row (space-separated):") distance_matrix = np.zeros((num_cities, num_cities)) for i in range(num_cities): row = list(map(int, input(f"Row {i + 1}: ").split())) distance_matrix[i] = row

Input PSO parameters num_particles = int(input("Enter the number of particles: ")) num_iterations = int(input("Enter the number of iterations: ")) w = float(input("Enter the inertia weight (w): ")) c1 = float(input("Enter the cognitive coefficient (c1): ")) c2 = float(input("Enter the social coefficient (c2): "))

```
    return distance_matrix, num_particles, num_iterations, w, c1, c2

# Get user input for the distance matrix and PSO parameters
distance_matrix, num_particles, num_iterations, w, c1, c2 = input_pso_parameters()

# Initialize PSO with the distance matrix and parameters
pso_tsp = PSO_TSP(distance_matrix, num_particles, num_iterations, w, c1, c2)

# Run PSO to find the shortest path
best_route, best_distance = pso_tsp.run()

print("\nBest route found:", best_route) print("Best
route distance:", best_distance)
```

Output:

```
Enter the number of cities: 4
Enter the distance matrix row by row (space-separated
Row 1: 0 5 10 15
Row 2: 5 0 20 30
Row 3: 30 10 0 5
Row 4: 5 10 15 0
Enter the number of particles: 50
Enter the number of iterations: 200
Enter the inertia weight (w): 0.7
Enter the cognitive coefficient (c1): 1.5
Enter the social coefficient (c2): 1.5
Iteration 1: Best Distance = 30.0
Iteration 2: Best Distance = 30.0
Iteration 3: Best Distance = 30.0
Iteration 4: Best Distance = 30.0
Iteration 5: Best Distance = 30.0
Iteration 6: Best Distance = 30.0
Iteration 7: Best Distance = 30.0
Iteration 8: Best Distance = 30.0
```

```
Iteration 185: Best Distance = 30.0
Iteration 186: Best Distance = 30.0
Iteration 187: Best Distance = 30.0
Iteration 188: Best Distance = 30.0
Iteration 189: Best Distance = 30.0
Iteration 190: Best Distance = 30.0
Iteration 191: Best Distance = 30.0
Iteration 192: Best Distance = 30.0
Iteration 193: Best Distance = 30.0
Iteration 194: Best Distance = 30.0
Iteration 195: Best Distance = 30.0
Iteration 196: Best Distance = 30.0
Iteration 197: Best Distance = 30.0
Iteration 198: Best Distance = 30.0
Iteration 199: Best Distance = 30.0
Iteration 200: Best Distance = 30.0
```

```
Best route found: [2 3 1 0]
Best route distance: 30.0
```

Program 3 Ant Colony Optimization for the Traveling Salesman Problem

Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:

LAB-4 7/11/24

→ Ant Colony Algorithm:-

Initialize parameters: numCities, numAnts, numIterations, alpha, beta, rho, Q

distance Matrix [numcities][numcities]
pheromone Matrix [numcities][numcities] = tau_0

For iteration = 1 to numIterations do:
Initialize tours for all ants :
for each ant k=1 to numAnts do :
Place ant k at a random starting city.
~~Initialize~~ Initialize empty tour for ant
K mark starting city as visited.

Repeat until all cities are visited :
For current city i, calculate transition probabilities
P_ij for all unvisited cities j :
P_ij = (pheromone matrix [i][j] ^α *
(1 / distance [i][j] ^β)) / sum of
probabilities for all unvisited cities.

choose next city based on P_ij (roulette wheel selection)
Add chosen city to tour of ant k.
mark chosen city as visited.
Move ant to chosen city.

calculate length L_k of completed tour for ant K

update best tour found (if applicable)
update pheromone levels:
    Evaporate pheromone on all paths:
        for each edge (i,j) in pheromone matrix:
            pheromoneMatrix[i][j] = (1 - rho) *
            pheromoneMatrix[i][j].

    Deposit pheromone based on tour of all ants:
        for each ant k = 1 to num ants:
            for each edge (i,j) in tour of ant k:
                pheromone Matrix [i][j] + Q / L-k.

Applications:

1) Travelling Sales man - TSP
2) Job assigning
3) Network design
4) Route optimization.

Purpose:-

1) Minimum cost
2) reduce travel time
3) improve efficiency.

O/P Recd

Execute

Code:

```
import numpy as np
import random

# Function to calculate the total distance of a given path def
calculate_total_distance(distance_matrix, path):
   total_distance = 0          for i in range(len(path) - 1):
total_distance += distance_matrix[path[i]][path[i + 1]]
   total_distance += distance_matrix[path[-1]][path[0]]   # Returning to the origin city
return total_distance

#     Function     to     perform     the     Ant     Colony     Optimization     def
ant_colony_optimization(distance_matrix, num_ants, num_iterations, alpha, beta, rho,
pheromone_initial):     num_cities = len(distance_matrix)

   # Initialize pheromone matrix with the initial pheromone value
pheromone = np.ones((num_cities, num_cities)) * pheromone_initial

   #    Initialize    the    best    solution
best_solution = None
   best_distance = float('inf')

   # Main ACO loop      for iteration in
range(num_iterations):
      # Ants' paths and their corresponding distances
paths = []
      distances = []

      # Generate solutions for each ant
for ant in range(num_ants):
```

```
        path = generate_path(distance_matrix, pheromone, alpha, beta)
total_distance = calculate_total_distance(distance_matrix, path)
paths.append(path)
        distances.append(total_distance)

        # Update the best solution if a new better one is found
if total_distance < best_distance:          best_solution =
path
          best_distance = total_distance

      # Update pheromones
      pheromone = update_pheromones(pheromone, paths, distances, rho, best_solution,
best_distance)

   return best_solution, best_distance

# Function to generate a solution (path) for an ant def
generate_path(distance_matrix, pheromone, alpha, beta):
num_cities = len(distance_matrix)
   path = [random.randint(0, num_cities - 1)] # Start at a random city    visited
= set(path)

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

      # Calculate the probabilities for all unvisited cities
for next_city in range(num_cities):          if next_city
not in visited:
          pheromone_strength = pheromone[current_city][next_city] ** alpha
distance_heuristic = (1.0 / distance_matrix[current_city][next_city]) ** beta
probabilities.append(pheromone_strength * distance_heuristic)          else:
          probabilities.append(0)
```

```
    # Normalize the probabilities
total_prob = sum(probabilities)
    probabilities = [p / total_prob for p in probabilities]

    # Choose the next city based on the calculated probabilities
next_city = np.random.choice(range(num_cities), p=probabilities)
path.append(next_city)
    visited.add(next_city)

  return path

# Function to update the pheromone matrix after each iteration def
update_pheromones(pheromone, paths, distances, rho, best_solution, best_distance):
  num_cities = len(pheromone)

  # Apply pheromone evaporation
  pheromone *= (1 - rho)

  # Deposit pheromones based on the paths and their distances
for path, dist in zip(paths, distances):
    for i in range(len(path) - 1):
      pheromone[path[i]][path[i + 1]] += 1.0 / dist
    pheromone[path[-1]][path[0]] += 1.0 / dist  # Returning to the origin city

  # Deposit more pheromone on the best path found so far    for i in
range(len(best_solution) - 1):    pheromone[best_solution[i]][best_solution[i
+ 1]] += 1.0 / best_distance
  pheromone[best_solution[-1]][best_solution[0]] += 1.0 / best_distance  # Returning to the origin
city

  return pheromone

# Input the distance matrix and parameters from the user print("Ant
Colony Application for Travelling Sales Man Problem")
```

```
num_cities = int(input("Enter the number of cities: "))
distance_matrix = [] print("Enter the distance matrix
(row by row):") for i in range(num_cities):      row =
list(map(int, input(f"Row {i+1}: ").split()))
    distance_matrix.append(row)

num_ants = int(input("Enter the number of ants: ")) num_iterations =
int(input("Enter the number of iterations: ")) alpha = float(input("Enter the value of
alpha (importance of pheromone): ")) beta = float(input("Enter the value of beta
(importance of heuristic information): ")) rho = float(input("Enter the evaporation
rate (rho): "))
pheromone_initial = float(input("Enter the initial pheromone value: "))

# Run the ACO algorithm
best_solution, best_distance = ant_colony_optimization(
    distance_matrix, num_ants, num_iterations, alpha, beta, rho, pheromone_initial
)

# Display the results print("Best Solution (Path):", list(map(int, best_solution)))  #
Fix for clean output print("Best Distance:", best_distance)
```

Output:

```
Ant Colony Application for Travelling Sales Man Problem
Enter the number of cities: 5
Enter the distance matrix (row by row):
Row 1: 0 5 10 15 20
Row 2: 10 0 15 20 30
Row 3: 5 20 0 15 20
Row 4: 30 15 5 0 30
Row 5: 20 5 10 15 20
Enter the number of ants: 10
Enter the number of iterations: 100
Enter the value of alpha (importance of pheromone): 1.0
Enter the value of beta (importance of heuristic information): 2.0
Enter the evaporation rate (rho): 0.5
Enter the initial pheromone value: 1.0
Best Solution (Path): [0, 4, 1, 3, 2]
Best Distance: 55
```

Program 4 Cuckoo Search (CS) Algorithm

Implement Cuckoo Search Algorithm for application Aerodynamics in engineering design.

Algorithm:

LAB - 5 14/11/24

=> Cuckoo Search Algorithm :-

1. Define the objective function, whethere we wanted to find the maximum or minimum solution.

2. Initialize parameters:
   No of Nests n.
   Find probability that nest is discovered and replaced.

3. Generate initial population of nest with random positions withing search space. (Xi) is the position

4. Evaluate fitness of each nest using objective function. f(X).

5. Generate New solutions by performing Ley flight.
   Calculate fitness of new solution, check weather the new solution is better than the previous one.

6. Repeat the iterations.

14/11/24

Application :- Engineering Design (Aerodynamics) → Implementation

Purpose :- to optimize design parameters and minimize aerodynamic drag.

o/p recs

Code :

```
import numpy as np

# Define the objective function: A simplified "drag function" that we aim to minimize def
drag_function(x):
    # x[0]: curvature, x[1]: width, x[2]: slope
    # A hypothetical drag equation (for demonstration purposes)
    return x[0]**2 + 2 * x[1]**2 + 3 * x[2]**2 + 4 * x[0] * x[1] - 2 * x[1] * x[2]

# Lévy flight function using numpy for Gamma and other computations
def gamma_function(x):     if x == 0.5:
        return np.sqrt(np.pi)   # Special case for gamma(1/2)
elif x == 1:
        return 1   # Special case for gamma(1)
elif x == 2:
        return 1   # Special case for gamma(2)
else:
        return np.math.factorial(int(x) - 1) if x.is_integer() else np.inf

def levy_flight(Lambda):      sigma = (gamma_function(1 + Lambda) *
np.sin(np.pi * Lambda / 2) /
            (gamma_function((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
u = np.random.randn() * sigma     v = np.random.randn()     step = u / abs(v) ** (1 / Lambda)
    return step

# Cuckoo Search Algorithm def cuckoo_search(n, iterations, pa,
lower_bound, upper_bound):
    # Initialize nests randomly     dim = 3  #
Number of design parameters
    nests = np.random.uniform(lower_bound, upper_bound, (n, dim))
```

```
    # Evaluate fitness of initial nests
    fitness = np.array([drag_function(nest) for nest in nests])
best_nest  =  nests[np.argmin(fitness)]          best_fitness  =
min(fitness)

    # Cuckoo  Search  main  loop
for    _    in    range(iterations):
for i in range(n):
            # Generate a new solution by Lévy flight
            step_size = levy_flight(1.5)
            new_nest = nests[i] + step_size * np.random.uniform(-1, 1, dim)
            new_nest  =  np.clip(new_nest,  lower_bound,  upper_bound)    #  Ensure  within  bounds
new_fitness = drag_function(new_nest)

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

        #  Abandon  a  fraction  of  the  worst  nests  and  create  new  ones
for i in range(int(pa * n)):
            nests[-(i    +    1)]    =    np.random.uniform(lower_bound,    upper_bound,    dim)
fitness[-(i + 1)] = drag_function(nests[-(i + 1)])

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

    return best_nest, best_fitness

# Gather user input for the algorithm
print("Welcome to the Aerodynamics Optimization using Cuckoo Search!")
```

```
n = int(input("Enter the number of nests (population size): ")) iterations =
int(input("Enter the number of iterations: ")) pa = float(input("Enter the
probability of abandonment (between 0 and 1): ")) lower_bound =
float(input("Enter the lower bound for the design parameters: ")) upper_bound =
float(input("Enter the upper bound for the design parameters: "))

# Run the Cuckoo Search algorithm
best_solution, best_drag_value = cuckoo_search(n, iterations, pa, lower_bound, upper_bound)

# Display the result print("\nOptimization Results:")
print("Best Solution (Design Parameters):", best_solution)
print("Best Drag Value:", best_drag_value)
```

Output :

```
Aerodynamics Optimization using Cuckoo Search
Enter the number of nests (population size): 10
Enter the number of iterations: 100
Enter the probability of abandonment (between 0 and 1): 0.25
Enter the lower bound for the design parameters: -10
Enter the upper bound for the design parameters: 10

Optimization Results:
Best Solution (Design Parameters): [-2.22259487 -9.7622218  -2.62606657]
Best Drag Value: -117.25613539786823
```

Program 5 Grey Wolf Optimizer (GWO)

Implement Grey Wolf Optimizer for path planning application in Robotics

Algorithm:

LAB - 6   28/11/24

→ Algorithm: Grey wolf Alg optimizer

1. Define the objective function $f(x)$. This function can be of maximization or minimization.

2. Initialize the input parameters.
   (i) n - no of wolfs
   (ii) I - no of iterations.

3. Initialize the population using randomly.
   Alpha: the best solution (leader)
   Beta: the second best solution (decision-making of alpha)
   delta: the third best solution (that guides the rest of the packs).
   omega: the remaining solutions. (they follow the above packs).

4. Evaluate the fitness:
   evaluate the fitness of each wolf based on the optimization / ~~fund~~ objective function.

5. Update the positions:
   Find the alpha, beta, and delta respectively from the first, second and third best solution from the fitness evaluated.

   $D_\alpha = |C_\alpha \cdot$ current position of α - position of omega $|$.

   New position (x) = current position of α - $A_\alpha \cdot D_\alpha$.

   Similarly compute new positions of beta and delta.

   New position = New position(α) + New position (beta) + New position (delta)

3

6. Repeat step 4 and 5 for all I iterations to obtain the best solution.

7. Output the best solution found.

Application:-

* data analysis
* Machine Learning
* Engineering Design
* Image processing

Code:

```
import numpy as np
import random

# Function to print student name and ID def
print_student_details():
    print("Chaitanya N INM22CS076\n")

# Environment: 2D grid def
get_grid_input():
    rows = int(input("Enter the number of rows in the grid: "))
    cols = int(input("Enter the number of columns in the grid: "))

    grid = []     print("Enter the grid values (0 for free space, -1 for
obstacles):")      for i in range(rows):           row = list(map(int,
input(f"Enter row {i+1}: ").split()))       grid.append(row)
```

```
    return grid

# Parameters max_iterations
= 100 population_size = 10

def is_valid_move(grid, x, y):
    """Check if a move is valid within the grid."""
    return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] != -1

def fitness(path, destination):      """Calculate fitness of a
path."""      if not path:           return float('inf')  # Invalid
paths have infinite fitness
    distance   =   len(path)     #   Length   of   the   path
end_point = path[-1]
    penalty = 0 if end_point == destination else 1000  # Penalty for not reaching the destination
return distance + penalty

def initialize_population(grid, source, destination, population_size):
    """Randomly initialize paths."""
population  =  []           for  _  in
range(population_size):
        path = [source]         current
= source           while current !=
destination:
            x, y = current
            #        Random        valid        move
possible_moves = [
                (x+1, y), (x-1, y), (x, y+1), (x, y-1)
            ]
            valid_moves = [move for move in possible_moves if is_valid_move(grid, *move) and
move not in path]           if not valid_moves:               break  # Dead end
            current = random.choice(valid_moves)
```

```
            path.append(current)
population.append(path)         return
population

def update_position(alpha, beta, delta, wolf, grid):
    """Update wolf position based on alpha, beta, delta wolves."""
new_path = []     for i in range(len(wolf)):
        if i < len(alpha) and is_valid_move(grid, *alpha[i]):
            new_path.append(alpha[i])         elif i < len(beta)
and is_valid_move(grid, *beta[i]):
            new_path.append(beta[i])          elif i < len(delta)
and                  is_valid_move(grid,                  *delta[i]):
new_path.append(delta[i])         else:            break
    return new_path

def       display_grid_with_path(grid,       path):
"""Display the grid with the path overlaid."""
path_set = set(path)     visual_grid = []     for i in
range(len(grid)):            row = []           for j in
range(len(grid[0])):            if (i, j) in path_set:
                row.append('*')    #  Mark  the  path
elif grid[i][j] == -1:
                row.append('X')      #   Represent   obstacles
else:
                row.append('.')  # Represent free spaces
        visual_grid.append(row)
return visual_grid


# Main GWO Algorithm def
gwo_path_planning():
print_student_details()
```

```
    # Get grid input from the user
    grid = get_grid_input()

    # Get start and destination points from user
    source = tuple(map(int, input("Enter the start point (x, y): ").split()))
    destination = tuple(map(int, input("Enter the destination point (x, y): ").split()))

    population = initialize_population(grid, source, destination, population_size)      for iteration in
range(max_iterations):         # Sort population by fitness
        population  =  sorted(population,  key=lambda  path:  fitness(path,  destination))
alpha, beta, delta = population[0], population[1], population[2]

        # Update positions        new_population = []        for wolf in
population:            new_path = update_position(alpha, beta, delta,
wolf,  grid)                       new_population.append(new_path)
population = new_population

    # Output the best path
    best_path   =   sorted(population,   key=lambda   path:   fitness(path,   destination))[0]
print(f"Best Path From {source} to {destination}: ", best_path)

    # Visualize the grid with the path
    visualized_grid = display_grid_with_path(grid, best_path)
    print("\nGrid showing the Best Path with stars representing the path and X representing
obstacles:")     for row in visualized_grid:
        print(' '.join(row))

# Call the function to run the program
gwo_path_planning()
```

Output:

```
Chaitanya N IBM22CS076

Enter the number of rows in the grid: 5
Enter the number of columns in the grid: 5
Enter the grid values (0 for free space, -1 for obstacles):
Enter row 1: 0 0 0 -1 0
Enter row 2: -1 -1 0 -1 0
Enter row 3: 0 0 0 0 0
Enter row 4: 0 -1 -1 -1 0
Enter row 5: 0 0 0 0 0
Enter the start point (x, y): 0 0
Enter the destination point (x, y): 4 4
Best Path From (0, 0) to (4, 4):  [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]

Grid showing the Best Path with stars representing the path and X representing obstacles:
* * * X .
X X * X .
. . * * *
. X X X *
. . . . *
```

Program 6 Parallel Cellular Algorithms and Programs

Implement Parallel Cellular Algorithms for application image processing edge detection . Algorithm:

LAB-7

Q] Parallel Cellular Algorithms and Programs:

Algorithm:

1. Initialize Grid and Population:
   a. Create a grid of size N x M
   b. Initialize each cell (i, j) with
      - random solution x - {i,j} in the solution space.
      - compute its fitness f(x-{i,j}) using objective function.

2. Repeat T iterations
   For each cell (i,j) in grid, in parallel:
   a. Identify the neighbours of the cell based on neighborhood structure.
      - Von Neumann (4 neighbours)
      - Moore (8 neighbour).
   b. Apply update rule:
      - Compare the current cell's solution with its neighbour's solution.
      - Update solution x - {i, j} based on best among neighbour.
   c. Recompute fitness ~~to~~ of updated solution f(x - {i, j}).

3. Track Best Solution.
   a. Maintain record of global best solution and its fitness

4. Check stopping condition.
   If maximum number of iterations is reached or convergence criteria is met, stop.

5. Output the best solution:
Return the global best solution and its fitness.

Applications :

1. Optimization problem : Resource allocation, job scheduling, travelling salesman.
2. Robotics - Path planning.

Code:

```
import cv2   import numpy as np   from
multiprocessing import Pool, cpu_count
from google.colab.patches import cv2_imshow # Import cv2_imshow for displaying images in
Colab

# Function to apply Sobel operator to a small image chunk
def apply_sobel(chunk):   # Sobel kernels   sobel_x =
np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])   sobel_y =
np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
# Pad chunk to handle edge cases   padded_chunk =
np.pad(chunk, ((1, 1), (1, 1)), mode='constant')   edge_chunk =
np.zeros_like(chunk)
# Apply Sobel operator   for i in range(1,
padded_chunk.shape[0] - 1):  for j in range(1,
padded_chunk.shape[1] - 1):
region = padded_chunk[i-1:i+2, j-1:j+2]   gx = np.sum(region * sobel_x)   gy =
np.sum(region * sobel_y)   edge_chunk[i-1, j-1] = min(255, np.sqrt(gx**2 +
gy**2)) # Gradient magnitude  return edge_chunk
```

Function to split the image into chunks def
split_image(image, num_chunks):
h, w = image.shape chunk_height
= h // num_chunks
# If image height is not divisible by num_chunks, ensure the last chunk gets the remaining rows
chunks = [image[i * chunk_height:(i + 1) * chunk_height] for i in range(num_chunks - 1)]
chunks.append(image[(num_chunks - 1) * chunk_height:]) # Add the last chunk with remaining rows return chunks

Function to combine chunks back into a single image def
combine_chunks(chunks):
return np.vstack(chunks)

Main function to process the image def
parallel_edge_detection(image_path, num_workers=None):
if num_workers is None: num_workers
= cpu_count() # Load image in
grayscale image =
cv2.imread(image_path,
cv2.IMREAD_GRAYSCALE)
if image is None:

raise FileNotFoundError(f"Image file not found: {image_path}")

Split the image into chunks for parallel processing chunks
= split_image(image, num_workers)
Process each chunk in parallel with
Pool(num_workers) as pool: processed_chunks =
pool.map(apply_sobel, chunks)
# Combine the processed chunks edge_image =
combine_chunks(processed_chunks) return image,
edge_image # Example usage if __name__ ==
"__main__":

print("Chaitanya N 1BM22CS076") input_image_path = "/content/image.jpeg" # Replace with your image path output_image_path = "output_edge_detected.jpg"
Run edge detection original_image, edge_detected_image = parallel_edge_detection(input_image_path)
Save the edge-detected image cv2.imwrite(output_image_path, edge_detected_image) # Combine original and edge-detected images side by side combined_image = np.hstack((original_image, edge_detected_image)) # Display the combined image in Colab cv2_imshow(combined_image) print(f"Edge-detected image saved as: {output_image_path}")

Output :

Program 7 Optimization via Gene Expression Algorithms

Implement Optimization via Gene Expression Algorithms for application

Algorithm

LAB-8

Optimization Via Gene Expression Algorithm:

1. Define the problem by objective function $f(x)$
2. Initialize parameters:
   - set population size
   - number of genes (G)
   - mutation rate (M)
   - crossover rate (C)
   - maximum generation (T)
3. Initialize population:
   Generate initial population P of random genetic sequences.
   Each genetic sequence consists of G genes
4. Evaluate Fitness:
   - For each genetic sequence in population:
     a. Translate genetic sequence into a solution $(x-1)$
     b. Compute fitness $f(x-1)$ using Objective func.
5. Repeat for each Generation ($t=1$ to T):
   a. selection:
   - select P genetic sequence for reproduction using selection mechanism.
   - Favour sequences with high fitness value.

   b. Crossover:
   - Randomly select pairs of genetic sequences from population.
   - with probability C, perform crossover to exchange points of genetic sequences to produce offspring.

c. Mutation:
- For each offspring, mutate genes with probability (M) to introduce variability.

d. Gene Expression.
- Translate the genetic sequence of offspring into functional solution (x-i)

e. evaluate fitness:
compute fitness f(x-i) of offspring sol.

f. Population replacement:
- Replace old population w offspring popula^n.

g. Track Best solution:
- Update global best sol & its fitness if improved.

6. Termination: Stop if max iterations reached or convergence criteria met.

7. Output best solution: return global best solution and its fitness.

Applications:
1. Optimization
2. path planning (robotics)
3. Pattern recognition (Data analysis)

Leela.B
18/12
seen

Code :

```
import numpy as np import random from
sklearn.datasets import make_classification from
sklearn.model_selection import train_test_split from
sklearn.metrics import accuracy_score
# 1. Define the Problem: Create a mathematical function to optimize (Pattern Recognition Task)

# For simplicity, we are using a classification dataset. def
create_synthetic_data():
# Create a simple synthetic classification dataset with 2 classes

X, y = make_classification(n_samples=100, n_features=5, n_classes=2, random_state=42) return
X, y
# 2. Initialize Parameters population_size =
20 num_genes = 5 # Number of features to
use mutation_rate = 0.1 crossover_rate =
0.7 num_generations = 100
# 3. Initialize Population: Randomly generate genetic sequences def
initialize_population(population_size, num_genes):
population = [] for _ in
range(population_size):
# Randomly initialize each gene between 0 and 1 (binary encoding of features)
genes = np.random.randint(2, size=num_genes)
population.append(genes) return
np.array(population)
# 4. Evaluate Fitness: Based on accuracy of model def
evaluate_fitness(population, X_train, X_test, y_train, y_test):
fitness_scores = [] for
individual in population:
# Here, the genes represent feature selection selected_features = [i for
i, gene in enumerate(individual) if gene == 1] if not selected_features:
# if no feature selected, it's an invalid solution fitness_scores.append(0)
continue
```

Train a simple classifier using the selected features

X_train_selected = X_train[:, selected_features]

X_test_selected = X_test[:, selected_features] # Train a basic classifier (e.g., Logistic Regression) from sklearn.linear_model import LogisticRegression clf = LogisticRegression() clf.fit(X_train_selected, y_train) # Make predictions and calculate accuracy y_pred = clf.predict(X_test_selected) accuracy = accuracy_score(y_test, y_pred) fitness_scores.append(accuracy)

return np.array(fitness_scores) # 5. Selection: Tournament Selection def select_parents(population, fitness_scores):

parents = [] for _ in range(len(population) // 2):

tournament_size = 3 selected = random.sample(list(zip(population, fitness_scores)), tournament_size) selected = sorted(selected, key=lambda x: x[1], reverse=True) parents.append(selected[0][0]) # Select the best individual parents.append(selected[1][0]) # Select the second best individual return np.array(parents)

6. Crossover: Single-point crossover def crossover(parents):

offspring = [] for i in range(0, len(parents), 2):

parent1 = parents[i] parent2 = parents[i + 1] if random.random() < crossover_rate:

crossover_point = random.randint(1, len(parent1) - 1) child1 = np.concatenate([parent1[:crossover_point], parent2[crossover_point:]]) child2 = np.concatenate([parent2[:crossover_point], parent1[crossover_point:]]) else:

child1, child2 = parent1.copy(), parent2.copy()

```
offspring.append(child1)
offspring.append(child2)  return
np.array(offspring)
# 7. Mutation: Flip bits with mutation rate
def mutate(offspring, mutation_rate):   for
i   in   range(len(offspring)):      for   j   in
range(len(offspring[i])):
if random.random() < mutation_rate:

offspring[i][j] = 1 - offspring[i][j] # Flip the gene  return
offspring
# 8. Gene Expression: Decode genetic sequences to functional solutions (feature selection in this
case)

# 9. Iterate: Repeat selection, crossover, mutation, and evaluation

def  gene_expression_algorithm(X_train,  X_test,  y_train,  y_test,  population_size,  num_genes,
num_generations, mutation_rate, crossover_rate):

population = initialize_population(population_size, num_genes)  for
generation in range(num_generations):
fitness_scores = evaluate_fitness(population, X_train, X_test, y_train, y_test)
parents   =   select_parents(population,   fitness_scores)      offspring   =
crossover(parents)  mutated_offspring = mutate(offspring, mutation_rate)
# Create the new population by replacing the old population with offspring  population
= mutated_offspring
# Print the best fitness score for each generation
print(f"Generation {generation + 1}: Best Fitness = {max(fitness_scores)}")

# Return the best solution (individual) from the final population  final_fitness_scores
= evaluate_fitness(population, X_train, X_test, y_train, y_test)  best_individual =
population[np.argmax(final_fitness_scores)]  return best_individual
# Main function to run the algorithm with user input  def
gwo_pattern_recognition():
```

Get user input for generations and population size generations
= int(input("Enter number of generations: ")) population_size =
int(input("Enter population size: "))
# Create synthetic data for pattern recognition

X, y = create_synthetic_data()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

Run the Gene Expression Algorithm

best_solution = gene_expression_algorithm(X_train, X_test, y_train, y_test, population_size, 5, generations, 0.1, 0.7) print(f"Best Feature Selection: {best_solution}") # Convert best_solution to feature selection selected_features = [i for i, gene in enumerate(best_solution) if gene == 1] print(f"Selected Features: {selected_features}")

Run the program if __name__ == "__main__":
print("Chaitanya N1BM22CS076") # Student Info
gwo_pattern_recognition()

Output:

```
Chaitanya N1BM22CS076
Enter number of generations: 50
Enter population size: 20
Generation 1: Best Fitness = 1.0
Generation 2: Best Fitness = 1.0
Generation 3: Best Fitness = 1.0
Generation 4: Best Fitness = 1.0
Generation 5: Best Fitness = 1.0
Generation 6: Best Fitness = 1.0
Generation 7: Best Fitness = 1.0
Generation 8: Best Fitness = 1.0
Generation 9: Best Fitness = 1.0
Generation 10: Best Fitness = 1.0
Generation 11: Best Fitness = 1.0
Generation 12: Best Fitness = 1.0
Generation 13: Best Fitness = 1.0
Generation 14: Best Fitness = 1.0
Generation 15: Best Fitness = 1.0
Generation 16: Best Fitness = 1.0
Generation 17: Best Fitness = 1.0
Generation 18: Best Fitness = 1.0
Generation 19: Best Fitness = 1.0
Generation 20: Best Fitness = 1.0
Generation 21: Best Fitness = 1.0
Generation 22: Best Fitness = 1.0
Generation 23: Best Fitness = 1.0
Generation 24: Best Fitness = 1.0
Generation 25: Best Fitness = 1.0
Generation 26: Best Fitness = 1.0
Generation 27: Best Fitness = 1.0
Generation 28: Best Fitness = 1.0
Generation 29: Best Fitness = 1.0
Generation 30: Best Fitness = 1.0
Generation 31: Best Fitness = 1.0
Generation 32: Best Fitness = 1.0
Generation 33: Best Fitness = 1.0
Generation 34: Best Fitness = 1.0
Generation 35: Best Fitness = 1.0
Generation 36: Best Fitness = 1.0
Generation 37: Best Fitness = 1.0
Generation 38: Best Fitness = 1.0
Generation 39: Best Fitness = 1.0
Generation 40: Best Fitness = 1.0
Generation 41: Best Fitness = 1.0
Generation 42: Best Fitness = 1.0
Generation 43: Best Fitness = 1.0
Generation 44: Best Fitness = 1.0
Generation 45: Best Fitness = 1.0
Generation 46: Best Fitness = 1.0
Generation 47: Best Fitness = 1.0
Generation 48: Best Fitness = 1.0
Generation 49: Best Fitness = 1.0
Generation 50: Best Fitness = 1.0
Best Feature Selection: [1 0 0 1 1]
Selected Features: [0, 3, 4]
```

: