VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems(23CS5BSBIS)

Submitted by

Yashraj Sinha (1BM22CS335)

in partial fulfilment 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 **Yashraj Sinha (1BM22CS335),** who is bonafide student of **B.M.S.College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering inComputer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academicrequirements of the above mentioned subject and thework prescribedfor the said degree.

Leelavathi .B Assistant Professor Department of CSE, BMSCE	Dr.Kavitha Sooda Professor &HOD Department of CSE, BMSCE

Index

Github Link: Yashraj_Github_Link

Program 1:

Problem Statement : Optimize the allocation of a portfolio using a Genetic Algorithm to maximize the Sharpe Ratio, balancing expected returns and risk. Ensure the total asset allocation adheres to a fixed budget constraint.

Algorithm:

Q) What is Genetic Algorithm?

→ A genetic algorithm (GA) is a search heuristic inspired by the principle of natural selection and genetics. It is used to solve optimization and search problems by mimicking the process of evolution. The basic idea is to evolve a population of candidates solutions to find the best solution to a given problem.

Key Components of Genetic Algorithm:

i) Population → A set of potential solution to the problem.

ii) Chromosome → A representation of a solution, typically encoded as a string of bits, numbers or characters.

iii) Fitness Function → A function that evaluates how good a solution is, typically mapping the chromosome to a numeric value representing its quality.

iv) Selection → The process of choosing the best candidate from the population based on their fitness scores.

v) Crossover → A genetic operator that combine two parent solution to create offspring solution

vi) Mutation → A genetic operator that introduces random

Date / /

vii) Replacement ⇒ The method used to determine how the new generation of solutions replaces the old one.

STEPS IN GENETIC ALGORITHM

1) Initialization → Generate an initial population of candidate solutions randomly

2) Evaluation → Compute the fitness of each candidate

3) Selection → Choose the fittest candidate to reproduce

4) Crossover → Combine pairs of candidates to create new offspring

5) Mutation → Randomly alter some offspring to maintain diversity.

6) Replacement → Form a new generation and repeat the evaluation.

Code:

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

# Function to calculate the total distance of a route
def calculate_total_distance(route, cities):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += np.linalg.norm(np.array(cities[route[i]]) - np.array(cities[route[i + 1]]))
    # Add distance to return to the starting point
    total_distance += np.linalg.norm(np.array(cities[route[-1]]) - np.array(cities[route[0]]))
    return total_distance

# Initialize population: Random routes
def initialize_population(pop_size, num_cities):
    population = []
    for _ in range(pop_size):
        route = list(range(num_cities))
        random.shuffle(route)
```

```
        population.append(route)
    return population

# Selection: Tournament selection
def selection(population, cities):
    fitness = []
    for route in population:
        fitness.append(1 / calculate_total_distance(route, cities))  # Inverse of distance (shorter is
better)
    total_fitness = sum(fitness)
    selected_parents = random.choices(population, weights=fitness, k=2)
    return selected_parents

# Crossover: Order Crossover (OX) method
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))

    child = [-1] * size
    child[start:end+1] = parent1[start:end+1]

    current_position = end + 1
    for i in range(size):
        if parent2[i] not in child:
            if current_position == size:
                current_position = 0
            child[current_position] = parent2[i]
            current_position += 1

    return child

# Mutation: Swap mutation
def mutate(route):
    idx1, idx2 = random.sample(range(len(route)), 2)
    route[idx1], route[idx2] = route[idx2], route[idx1]
    return route

# Main Genetic Algorithm function
def genetic_algorithm(cities, pop_size, generations, mutation_rate, crossover_rate):
    num_cities = len(cities)
    population = initialize_population(pop_size, num_cities)
    best_route = None
    best_distance = float('inf')

    for generation in range(generations):
        new_population = []

        for _ in range(pop_size // 2):
            parent1, parent2 = selection(population, cities)
            if random.random() < crossover_rate:
                child1 = crossover(parent1, parent2)
                child2 = crossover(parent2, parent1)
            else:
```

```
            child1, child2 = parent1, parent2  # No crossover, just parents

        if random.random() < mutation_rate:
            child1 = mutate(child1)
        if random.random() < mutation_rate:
            child2 = mutate(child2)

        new_population.extend([child1, child2])

    population = new_population

    # Evaluate the best solution so far
    for route in population:
        distance = calculate_total_distance(route, cities)
        if distance < best_distance:
            best_route = route
            best_distance = distance

    print(f"Generation {generation + 1}: Best Distance = {best_distance}")

return best_route, best_distance

# Visualization function
def plot_route(route, cities):
    route_cities = [cities[i] for i in route] + [cities[route[0]]]  # Complete the loop
    route_cities = np.array(route_cities)
    plt.plot(route_cities[:, 0], route_cities[:, 1], marker='o')
    plt.scatter(route_cities[:, 0], route_cities[:, 1], color='red')
    plt.show()

# Example usage
if __name__ == "__main__":
    # Define the coordinates of cities (e.g., 5 cities)
    cities = np.array([
        [0, 0],  # City 1
        [1, 2],  # City 2
        [4, 0],  # City 3
        [4, 3],  # City 4
        [2, 4],  # City 5
    ])

    # Parameters
    population_size = 50
    generations = 500
    mutation_rate = 0.1
    crossover_rate = 0.9

    # Run Genetic Algorithm
    best_route, best_distance = genetic_algorithm(cities, population_size, generations, mutation_rate,
crossover_rate)

    # Output the best solution found
    print(f"Best Route: {best_route}")
```

```
print(f"Best Distance: {best_distance}")

# Plot the best route
plot_route(best_route, cities)
```

Output :

```
Generation 466: Best Distance = 13.70820393249937
Generation 467: Best Distance = 13.70820393249937
Generation 468: Best Distance = 13.70820393249937
Generation 469: Best Distance = 13.70820393249937
Generation 470: Best Distance = 13.70820393249937
Generation 471: Best Distance = 13.70820393249937
Generation 472: Best Distance = 13.70820393249937
Generation 473: Best Distance = 13.70820393249937
Generation 474: Best Distance = 13.70820393249937
Generation 475: Best Distance = 13.70820393249937
Generation 476: Best Distance = 13.70820393249937
Generation 477: Best Distance = 13.70820393249937
Generation 478: Best Distance = 13.70820393249937
Generation 479: Best Distance = 13.70820393249937
Generation 480: Best Distance = 13.70820393249937
Generation 481: Best Distance = 13.70820393249937
Generation 482: Best Distance = 13.70820393249937
Generation 483: Best Distance = 13.70820393249937
Generation 484: Best Distance = 13.70820393249937
Generation 485: Best Distance = 13.70820393249937
Generation 486: Best Distance = 13.70820393249937
Generation 487: Best Distance = 13.70820393249937
Generation 488: Best Distance = 13.70820393249937
Generation 489: Best Distance = 13.70820393249937
Generation 490: Best Distance = 13.70820393249937
Generation 491: Best Distance = 13.70820393249937
Generation 492: Best Distance = 13.70820393249937
Generation 493: Best Distance = 13.70820393249937
Generation 494: Best Distance = 13.70820393249937
Generation 495: Best Distance = 13.70820393249937
Generation 496: Best Distance = 13.70820393249937
Generation 497: Best Distance = 13.70820393249937
Generation 498: Best Distance = 13.70820393249937
Generation 499: Best Distance = 13.70820393249937
Generation 500: Best Distance = 13.70820393249937
Best Route: [4, 3, 2, 0, 1]
Best Distance: 13.70820393249937
```

Program 2:

Problem Statement : Implement a Particle Swarm Optimization (PSO) algorithm to minimize benchmark functions, such as the Rastrigin and Sphere functions, by optimizing their input parameters. The goal is to find the global minimum while efficiently exploring the solution space using swarm intelligence.

Algorithm :

v) Algorithm

(a) Define the fitness function $f(x)$.

(b) Initialise parameters:

→ set num_particles, num_iterations, inertia_weight, cognitive_coefficient, social_coefficient

(c) Initialise particles:

→ Create an array of particles
→ For each particle:
↳ Randomly ~~in~~ initialise position
↳ Randomly initialise velocity
↳ Set best position to position
↳ Evaluate and store fitness

(d) Evaluate fitness for each particle

(e) Update velocities and positions for each particle based on personal and global bests.

(f) Iterate step 4 and step 5 for num_iterations.

(g) Output the best solution found during the iterations.

Code:

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

# Sphere function (fitness function)
def sphere_function(position):
    return np.sum(position**2)

# Parameters
num_particles = 30  # Number of particles
num_iterations = 100  # Number of iterations
dim = 2  # Dimensionality of the problem
bounds = [-10, 10]  # Search space bounds
inertia_weight = 0.7  # w
cognitive_coefficient = 1.5  # c1
social_coefficient = 1.5  # c2
tolerance = 1e-6  # Stopping tolerance for fitness

# Initialize particles
class Particle:
    def __init__(self):
        self.position = np.random.uniform(bounds[0], bounds[1], dim)  # Random position
        self.velocity = np.random.uniform(-1, 1, dim)  # Random velocity
        self.best_position = np.copy(self.position)  # Personal best position
        self.best_fitness = sphere_function(self.position)  # Personal best fitness
        self.fitness = self.best_fitness  # Current fitness

    def update_velocity(self, global_best_position):
        r1, r2 = np.random.rand(dim), np.random.rand(dim)
        cognitive_term = cognitive_coefficient * r1 * (self.best_position - self.position)
        social_term = social_coefficient * r2 * (global_best_position - self.position)
        self.velocity = inertia_weight * self.velocity + cognitive_term + social_term

    def update_position(self):
        self.position += self.velocity
        self.position = np.clip(self.position, bounds[0], bounds[1])  # Keep within bounds
        self.fitness = sphere_function(self.position)
        if self.fitness < self.best_fitness:  # Update personal best
            self.best_fitness = self.fitness
            self.best_position = np.copy(self.position)

# PSO implementation
def particle_swarm_optimization():
    particles = [Particle() for _ in range(num_particles)]
    global_best_position = particles[0].best_position
    global_best_fitness = particles[0].best_fitness

    fitness_history = []

    # Update global best from the initial population
    for particle in particles:
        if particle.best_fitness < global_best_fitness:
```

```
            global_best_fitness = particle.best_fitness
            global_best_position = np.copy(particle.best_position)

    for iteration in range(num_iterations):
        for particle in particles:
            particle.update_velocity(global_best_position)
            particle.update_position()

            # Update global best
            if particle.best_fitness < global_best_fitness:
                global_best_fitness = particle.best_fitness
                global_best_position = np.copy(particle.best_position)

        fitness_history.append(global_best_fitness)  # Track global best fitness

        # Early stopping if fitness reaches tolerance
        if global_best_fitness <= tolerance:
            print(f"Converged at iteration {iteration}")
            break

    return global_best_position, global_best_fitness, fitness_history

# Run the PSO algorithm
best_position, best_fitness, fitness_history = particle_swarm_optimization()

# Print the results
print("Best position found:", best_position)
print("Best fitness achieved:", best_fitness)

# Plot fitness over iterations
plt.plot(fitness_history)
plt.title("Fitness Over Iterations (PSO on Sphere Function)")
plt.xlabel("Iteration")
plt.ylabel("Fitness (Objective Function Value)")
plt.grid()
plt.show()
```

Output:

```
input
Converged at iteration 37
Best position found: [ 0.00080558 -0.00058912]
Best fitness achieved: 9.960226882351884e-07
/home/main.py:87: UserWarning: Matplotlib is currently using agg, which is a non-GUI backend, so cannot show the figure.
  plt.show()
```

Program 3

Problem Statement : Implement an Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP), where the goal is to find the shortest possible path that visits all cities exactly once and returns to the starting city. The algorithm should utilize pheromone trails and heuristic information to guide the search efficiently.

Algorithm :

~~Pseudo code~~ Algorithm

Initialize pheromone matrix τ and distance matrix D
Set parameters : α, β, ρ, max-iterations, number of ants M

For each iteration :
    For each ant :
        Initialize a path starting from random city
        Repeat until all cities are visited :
            Select the next city based on pheromone and add distance (using α and β).
    End For

    Evaluate the total distance of each path

    Update pheromone matrix by evaporating old pheromones and adding new pheromones.
    Track the shortest path so far

Return the best path and its distance.

Neela.B
7/11/24

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import euclidean

# Define the problem: coordinates of cities
cities = np.array([
    [0, 0], [2, 3], [5, 2], [6, 6], [8, 3],
    [1, 5], [4, 7], [7, 8], [9, 5], [3, 1]
])
num_cities = len(cities)

# Parameters
num_ants = 10
num_iterations = 100
alpha = 1  # Importance of pheromone
beta = 2  # Importance of heuristic (1/distance)
rho = 0.5  # Pheromone evaporation rate
initial_pheromone = 1.0

# Distance matrix
distances = np.array([[euclidean(cities[i], cities[j]) for j in range(num_cities)] for i in
range(num_cities)])

# Heuristic information (1 / distance), avoiding division by zero
heuristics = np.zeros_like(distances)  # Initialize as zero
for i in range(num_cities):
    for j in range(num_cities):
        if i != j:
            heuristics[i, j] = 1 / distances[i, j]  # Only calculate for non-diagonal elements
        else:
            heuristics[i, j] = 0  # Set diagonal to zero (no heuristic for the same city)

# Pheromone matrix
pheromones = np.full((num_cities, num_cities), initial_pheromone)

# Ant class
class Ant:
    def __init__(self):
        self.visited = []
        self.total_distance = 0

    def choose_next_city(self, current_city):
        probabilities = []
        for city in range(num_cities):
            if city not in self.visited:
                pheromone = pheromones[current_city][city] ** alpha
                heuristic = heuristics[current_city][city] ** beta
                probabilities.append(pheromone * heuristic)
            else:
```

```
                probabilities.append(0)

        # Convert to numpy array for easier manipulation
        probabilities = np.array(probabilities)

        # Check if all probabilities are zero, avoid NaN
        if probabilities.sum() == 0:
            unvisited_cities = [city for city in range(num_cities) if city not in self.visited]
            return np.random.choice(unvisited_cities)

        # Normalize probabilities to make sure they sum to 1
        probabilities /= probabilities.sum()
        return np.random.choice(range(num_cities), p=probabilities)

    def complete_tour(self):
        # Complete the tour by returning to the start city
        self.total_distance += distances[self.visited[-1]][self.visited[0]]
        self.visited.append(self.visited[0])

# ACO implementation
def ant_colony_optimization():
    global pheromones
    best_solution = None
    best_distance = float('inf')
    best_history = []

    for iteration in range(num_iterations):
        all_ants = []

        # Each ant constructs a solution
        for _ in range(num_ants):
            ant = Ant()
            current_city = np.random.randint(0, num_cities)  # Start at a random city
            ant.visited.append(current_city)

            while len(ant.visited) < num_cities:
                next_city = ant.choose_next_city(current_city)
                ant.total_distance += distances[current_city][next_city]
                ant.visited.append(next_city)
                current_city = next_city

            # Complete the tour
            ant.complete_tour()
            all_ants.append(ant)

            # Update global best
            if ant.total_distance < best_distance:
                best_distance = ant.total_distance
                best_solution = ant.visited

        # Update pheromone trails
        pheromones *= (1 - rho)  # Evaporation
        for ant in all_ants:
```

```
            for i in range(num_cities):
                from_city = ant.visited[i]
                to_city = ant.visited[i + 1] if i + 1 < len(ant.visited) else ant.visited[0]
                pheromones[from_city][to_city] += 1 / ant.total_distance

        # Track the best distance in history
        best_history.append(best_distance)
        print(f"Iteration {iteration + 1}, Best Distance: {best_distance}")

    return best_solution, best_distance, best_history

# Run the ACO algorithm
best_solution, best_distance, best_history = ant_colony_optimization()

# Print the results
print("\nBest route found:", best_solution)
print("Shortest distance:", best_distance)

# Plot the best route
route_cities = cities[best_solution]
plt.figure(figsize=(8, 6))
plt.plot(route_cities[:, 0], route_cities[:, 1], 'o-', label='Best Route')
plt.title("Best Route Found by ACO")
plt.xlabel("X Coordinate")
plt.ylabel("Y Coordinate")
plt.legend()
plt.grid()
plt.show()

# Plot the convergence
plt.figure()
plt.plot(best_history)
plt.title("ACO Convergence")
plt.xlabel("Iteration")
plt.ylabel("Shortest Distance")
plt.grid()
plt.show()
```

Output

```
Iteration 70, Best Distance: 28.321549034227672
Iteration 71, Best Distance: 28.321549034227672
Iteration 72, Best Distance: 28.321549034227672
Iteration 73, Best Distance: 28.321549034227672
Iteration 74, Best Distance: 28.321549034227672
Iteration 75, Best Distance: 28.321549034227672
Iteration 76, Best Distance: 28.321549034227672
Iteration 77, Best Distance: 28.321549034227672
Iteration 78, Best Distance: 28.321549034227672
Iteration 79, Best Distance: 28.321549034227672
Iteration 80, Best Distance: 28.321549034227672
Iteration 81, Best Distance: 28.321549034227672
Iteration 82, Best Distance: 28.321549034227672
Iteration 83, Best Distance: 28.321549034227672
Iteration 84, Best Distance: 28.321549034227672
Iteration 85, Best Distance: 28.321549034227672
Iteration 86, Best Distance: 28.321549034227672
Iteration 87, Best Distance: 28.321549034227672
Iteration 88, Best Distance: 28.321549034227672
Iteration 89, Best Distance: 28.321549034227672
Iteration 90, Best Distance: 28.321549034227672
Iteration 91, Best Distance: 28.321549034227672
Iteration 92, Best Distance: 28.321549034227672
Iteration 93, Best Distance: 28.321549034227672
Iteration 94, Best Distance: 28.321549034227672
Iteration 95, Best Distance: 28.321549034227672
Iteration 96, Best Distance: 28.321549034227672
Iteration 97, Best Distance: 28.321549034227672
Iteration 98, Best Distance: 28.321549034227672
Iteration 99, Best Distance: 28.321549034227672
Iteration 100, Best Distance: 28.321549034227672

Best route found: [7, 3, 6, 5, 1, 0, 9, 2, 4, 8, 7]
Shortest distance: 28.321549034227672
```

Program 4

Problem Statement : Implement the Cuckoo Search Algorithm for Smart Traffic Controli.

Algorithm :

1/11/24 Lab - 4 Date / / /

Cuckoo Search for Smart Traffic Control

* Algorithm :

1) Initialize the population of nests (solution) randomly within the search space

2) Evaluate the fitness of each nest based on optimization function.

3) Set the no. of nests (N), the probability of discovery (p) and the maximum no. of iteration (maxIter).

4) While (stopping condition not met and iteration < maxIter):
   Generate a new solution (nest) for each cuckoo using Levy flights :
   ~~- For each nest i, generate a new candidate solution using Levy flight :~~
   new_solution_i = current_solution_i + Levy_flight()

   Evaluate the fitness of the new solution
   If new solution is better than the ~~old~~ ~~one~~ ~~than~~ current solution then replace the old nest with the new one.

   Abandon the worst nests (solutions)

5) Return the best solution

6) End.

Code:

```
import numpy as np
import random

# Define the problem: Traffic control optimization
def congestion_function(signal_timings):
    """
    Simulated function to measure traffic congestion.
    Lower values indicate better traffic flow.
    """
```

```
    return np.sum((signal_timings - ideal_timings)**2)

# Parameters for the problem
num_signals = 4  # Number of traffic signals
ideal_timings = np.array([30, 40, 50, 60])  # Ideal timings for minimal congestion

# Cuckoo Search Parameters
num_nests = 10  # Number of nests (solutions)
num_iterations = 100  # Number of iterations
pa = 0.25  # Discovery probability
bounds = [(10, 90)] * num_signals  # Timing bounds for each signal

# Lévy flight function
def levy_flight(Lambda=1.5):
    sigma = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
             (np.math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = np.random.normal(0, sigma, size=num_signals)
    v = np.random.normal(0, 1, size=num_signals)
    step = u / abs(v)**(1 / Lambda)
    return step

# Initialize nests (random solutions)
def initialize_nests(num_nests, bounds):
    return np.array([[random.uniform(low, high) for low, high in bounds] for _ in range(num_nests)])

# Replace worst nests
def replace_worst_nests(nests, fitness, pa, bounds):
    num_replacements = int(pa * len(nests))
    worst_indices = np.argsort(fitness)[-num_replacements:]
    for idx in worst_indices:
        nests[idx] = np.array([random.uniform(low, high) for low, high in bounds])
    return nests

# Cuckoo Search Algorithm
def cuckoo_search():
    nests = initialize_nests(num_nests, bounds)
    best_nest = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        # Fitness evaluation
        fitness = np.array([congestion_function(nest) for nest in nests])

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

```
        # Generate new solutions via Lévy flights
        new_nests = np.array([nest + levy_flight() for nest in nests])
        new_nests = np.clip(new_nests, [low for low, high in bounds], [high for low, high in bounds])

        # Evaluate new fitness
        new_fitness = np.array([congestion_function(nest) for nest in new_nests])

        # Select better solutions
        for i in range(num_nests):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]

        # Abandon worst nests
        nests = replace_worst_nests(nests, fitness, pa, bounds)

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

    return best_nest, best_fitness

# Run the Cuckoo Search algorithm
best_solution, best_fitness = cuckoo_search()

# Output the results
print("\nOptimal Signal Timings:", best_solution)
print("Minimal Congestion Measure:", best_fitness)
```

Output:

```
Iteration 64, Best Fitness: 1.7906753674151423
Iteration 65, Best Fitness: 1.7906753674151423
Iteration 66, Best Fitness: 0.7433790320396539
Iteration 67, Best Fitness: 0.7433790320396539
Iteration 68, Best Fitness: 0.7433790320396539
Iteration 69, Best Fitness: 0.7433790320396539
Iteration 70, Best Fitness: 0.7433790320396539
Iteration 71, Best Fitness: 0.7433790320396539
Iteration 72, Best Fitness: 0.7433790320396539
Iteration 73, Best Fitness: 0.7433790320396539
Iteration 74, Best Fitness: 0.7433790320396539
Iteration 75, Best Fitness: 0.7433790320396539
Iteration 76, Best Fitness: 0.7433790320396539
Iteration 77, Best Fitness: 0.7433790320396539
Iteration 78, Best Fitness: 0.7433790320396539
Iteration 79, Best Fitness: 0.3113637489279607
Iteration 80, Best Fitness: 0.3113637489279607
Iteration 81, Best Fitness: 0.3113637489279607
Iteration 82, Best Fitness: 0.3113637489279607
Iteration 83, Best Fitness: 0.3113637489279607
Iteration 84, Best Fitness: 0.3113637489279607
Iteration 85, Best Fitness: 0.3113637489279607
Iteration 86, Best Fitness: 0.3113637489279607
Iteration 87, Best Fitness: 0.3113637489279607
Iteration 88, Best Fitness: 0.3113637489279607
Iteration 89, Best Fitness: 0.3113637489279607
Iteration 90, Best Fitness: 0.3113637489279607
Iteration 91, Best Fitness: 0.3113637489279607
Iteration 92, Best Fitness: 0.3113637489279607
Iteration 93, Best Fitness: 0.3113637489279607
Iteration 94, Best Fitness: 0.3113637489279607
Iteration 95, Best Fitness: 0.3113637489279607
Iteration 96, Best Fitness: 0.3113637489279607
Iteration 97, Best Fitness: 0.3113637489279607
Iteration 98, Best Fitness: 0.3113637489279607
Iteration 99, Best Fitness: 0.3113637489279607
Iteration 100, Best Fitness: 0.3113637489279607

Optimal Signal Timings: [30.28550549 40.28899424 49.79466885 60.32275659]
Minimal Congestion Measure: 0.3113637489279607
```

Program 5

Problem Statement : Implement the Grey Wolf Optimizer (GWO) to optimize the Placement of Solar Panels to maximize the output.

Algorithm :

Algorithm

i) We are optimizing the tilt angle and orientation angle of the solar panel to maximize energy output. (Objective)

ii) Parameters => Set the no. of wolves and iterations.

iii) Randomly generate initial positions for each wolf in the search space for tilt and orientation angles.

iv) Calculate the energy output ~~and~~ according to the fitness function

Date / /

v) ~~Evaluate~~ Adjust each wolves position based on alpha, beta and delta wolves.

vi) Repeat the evaluation & positions based on no. of iterations.

vii) Track and print the best position found and the corresponding energy output.

21/11/24

Code:

```
import numpy as np

class GreyWolfOptimizer:
    def __init__(self, func, dim, num_wolves, max_iter):
        self.func = func  # The objective function to maximize (solar energy)
        self.dim = dim  # Dimension of the search space (tilt and orientation)
        self.num_wolves = num_wolves  # Number of wolves in the population
        self.max_iter = max_iter  # Maximum number of iterations

        # Initialize the position of wolves randomly
        self.wolves_pos = np.random.uniform(-90, 90, (self.num_wolves, self.dim))  # Angles between -
90 and 90
        self.alpha_pos = np.zeros(self.dim)
        self.beta_pos = np.zeros(self.dim)
        self.delta_pos = np.zeros(self.dim)
        self.alpha_score = float("-inf")
        self.beta_score = float("-inf")
        self.delta_score = float("-inf")

    def fitness(self, pos):
        """Evaluate the fitness of a position (solar energy)"""
        tilt_angle = pos[0]
        orientation_angle = pos[1]

        # Simulate a simple energy function based on tilt and orientation.
        # This is just a mock function that peaks at tilt=30 and orientation=0 degrees
        energy_output = np.sin(np.radians(tilt_angle)) * np.cos(np.radians(orientation_angle)) + 1

        return energy_output  # Higher is better

    def update_positions(self, a):
        """Update the positions of wolves"""
        for i in range(self.num_wolves):
            r1 = np.random.random(self.dim)
            r2 = np.random.random(self.dim)

            A1 = 2 * a * r1 - a  # Coefficient for alpha wolf
            C1 = 2 * r2  # Coefficient for alpha wolf

            A2 = 2 * a * r1 - a  # Coefficient for beta wolf
            C2 = 2 * r2  # Coefficient for beta wolf

            A3 = 2 * a * r1 - a  # Coefficient for delta wolf
            C3 = 2 * r2  # Coefficient for delta wolf

            # Update the position of the current wolf based on the alpha, beta, and delta wolves
            D_alpha = np.abs(C1 * self.alpha_pos - self.wolves_pos[i, :])
            D_beta = np.abs(C2 * self.beta_pos - self.wolves_pos[i, :])
            D_delta = np.abs(C3 * self.delta_pos - self.wolves_pos[i, :])
```

```
            X1 = self.alpha_pos - A1 * D_alpha
            X2 = self.beta_pos - A2 * D_beta
            X3 = self.delta_pos - A3 * D_delta

            # Update the position of the current wolf
            self.wolves_pos[i, :] = (X1 + X2 + X3) / 3

    def optimize(self):
        """Run the Grey Wolf Optimizer"""
        for t in range(self.max_iter):
            a = 2 - t * (2 / self.max_iter)  # Coefficient decreases over time

            for i in range(self.num_wolves):
                fitness_score = self.fitness(self.wolves_pos[i, :])

                # Update the alpha, beta, and delta wolves
                if fitness_score > self.alpha_score:
                    self.alpha_score = fitness_score
                    self.alpha_pos = self.wolves_pos[i, :]
                elif fitness_score > self.beta_score:
                    self.beta_score = fitness_score
                    self.beta_pos = self.wolves_pos[i, :]
                elif fitness_score > self.delta_score:
                    self.delta_score = fitness_score
                    self.delta_pos = self.wolves_pos[i, :]

            # Update the positions of the wolves
            self.update_positions(a)

            print(f"Iteration {t+1}/{self.max_iter}, Best Solar Energy: {self.alpha_score}")

        return self.alpha_pos, self.alpha_score

# Set up the optimizer for Solar Panel Placement
dim = 2  # Two parameters to optimize: tilt angle and orientation angle
num_wolves = 15  # Number of wolves
max_iter = 50  # Number of iterations

# Initialize the Grey Wolf Optimizer
gwo = GreyWolfOptimizer(func=lambda x: x[0] * x[1], dim=dim, num_wolves=num_wolves,
max_iter=max_iter)

# Optimize the placement of the solar panel
best_position, best_score = gwo.optimize()

print("Best Placement (Tilt Angle, Orientation Angle):", best_position)
print("Best Solar Energy Output:", best_score)
```

Output:

```
Iteration 17/50, Best Solar Energy: 1.9964231475128704
Iteration 18/50, Best Solar Energy: 1.9964231475128704
Iteration 19/50, Best Solar Energy: 1.9964231475128704
Iteration 20/50, Best Solar Energy: 1.9964231475128704
Iteration 21/50, Best Solar Energy: 1.9964231475128704
Iteration 22/50, Best Solar Energy: 1.9964231475128704
Iteration 23/50, Best Solar Energy: 1.9964231475128704
Iteration 24/50, Best Solar Energy: 1.9964231475128704
Iteration 25/50, Best Solar Energy: 1.9964231475128704
Iteration 26/50, Best Solar Energy: 1.9964231475128704
Iteration 27/50, Best Solar Energy: 1.9964231475128704
Iteration 28/50, Best Solar Energy: 1.9964231475128704
Iteration 29/50, Best Solar Energy: 1.9964231475128704
Iteration 30/50, Best Solar Energy: 1.9964231475128704
Iteration 31/50, Best Solar Energy: 1.9964231475128704
Iteration 32/50, Best Solar Energy: 1.9964231475128704
Iteration 33/50, Best Solar Energy: 1.9964231475128704
Iteration 34/50, Best Solar Energy: 1.9964231475128704
Iteration 35/50, Best Solar Energy: 1.9964231475128704
Iteration 36/50, Best Solar Energy: 1.9964231475128704
Iteration 37/50, Best Solar Energy: 1.9964231475128704
Iteration 38/50, Best Solar Energy: 1.9978032416582416
Iteration 39/50, Best Solar Energy: 1.9978032416582416
Iteration 40/50, Best Solar Energy: 1.9978032416582416
Iteration 41/50, Best Solar Energy: 1.9978032416582416
Iteration 42/50, Best Solar Energy: 1.9978032416582416
Iteration 43/50, Best Solar Energy: 1.9978032416582416
Iteration 44/50, Best Solar Energy: 1.9978032416582416
Iteration 45/50, Best Solar Energy: 1.9978032416582416
Iteration 46/50, Best Solar Energy: 1.9978032416582416
Iteration 47/50, Best Solar Energy: 1.9978032416582416
Iteration 48/50, Best Solar Energy: 1.9978032416582416
Iteration 49/50, Best Solar Energy: 1.9978032416582416
Iteration 50/50, Best Solar Energy: 1.9978032416582416
Best Placement (Tilt Angle, Orientation Angle): [2041.2490479  1469.93892641]
Best Solar Energy Output: 1.9978032416582416
```

Program 6

Problem Statement : Parallel Cellular Algorithm for Traffic Flow Management.

Algorithm :

Algorithm

1> Initialize the traffic grid.

2> Randomly assign traffic ~~dest~~ densities to each intersection.

3> Define fitness function to evaluate total congestion

4> For each iteration:

ⓐ Evaluate the fitness of the current grid.
ⓑ For each iteration, update its state based on the traffic density.
ⓒ Update grid in parallel
ⓓ Track the best configuration so far.

5> Repeat the process for a set of number of iterations or until the best coverage.

6> Output the best grid and corresponding fitness value.

Code:

```
import numpy as np
import random
from multiprocessing import Pool

# Step 1: Initialize the traffic grid
def initialize_grid(rows, cols):
    """Create a grid with given rows and columns initialized to zero."""
    return np.zeros((rows, cols), dtype=int)

# Step 2: Assign random traffic densities
def assign_random_traffic(grid):
    """Randomly assign traffic densities (values between 1 to 10)."""
    rows, cols = grid.shape
    for i in range(rows):
        for j in range(cols):
            grid[i][j] = random.randint(1, 10)
    return grid

# Step 3: Define the fitness function
def fitness_function(grid):
    """Calculate fitness as the total congestion (sum of traffic densities)."""
    return np.sum(grid)

# Step 4: Update grid based on traffic density
def update_cell(cell_value):
    """Update a single grid cell's value based on some traffic rules."""
    if cell_value > 8:  # Simulating high congestion reduction
        return cell_value - 2
    elif cell_value < 3:  # Simulating increased density for low values
        return cell_value + 1
    return cell_value  # No change for moderate traffic

def update_grid_parallel(grid):
    """Update the grid in parallel."""
    rows, cols = grid.shape
    flat_grid = grid.flatten()
    with Pool() as pool:
        updated_flat_grid = pool.map(update_cell, flat_grid)
    return np.array(updated_flat_grid).reshape(rows, cols)

# Step 5: Main Traffic Flow Optimization Algorithm
def traffic_flow_optimization(rows, cols, iterations):
    """
    Perform traffic flow optimization for a given grid size and iterations.
    """
    # Step 1: Initialize grid
    grid = initialize_grid(rows, cols)
    best_grid = None
    best_fitness = float('inf')
```

```
    # Step 2: Assign random traffic densities
    grid = assign_random_traffic(grid)

    # Step 3 and 4: Iterate and optimize
    for iteration in range(iterations):
        print(f"\nIteration {iteration + 1}:")

        # Evaluate current fitness
        current_fitness = fitness_function(grid)
        print(f"Current Fitness: {current_fitness}")

        # Track the best configuration
        if current_fitness < best_fitness:
            best_fitness = current_fitness
            best_grid = grid.copy()
            print("Best grid updated!")

        # Update grid in parallel
        grid = update_grid_parallel(grid)
        print("Grid updated.")

    # Step 6: Output the best grid and fitness value
    print("\nOptimization Complete!")
    print(f"Best Fitness Value: {best_fitness}")
    print("Best Grid Configuration:")
    print(best_grid)

# Step 6: Run the algorithm
if __name__ == "__main__":
    rows = 5  # Number of rows in grid
    cols = 5  # Number of columns in grid
    iterations = 10  # Number of iterations to perform

    traffic_flow_optimization(rows, cols, iterations)
```

Output:

```
Current Fitness: 141
Grid updated.

Iteration 9:
Current Fitness: 141
Grid updated.

Iteration 10:
Current Fitness: 141
Grid updated.

Optimization Complete!
Best Fitness Value: 139
Best Grid Configuration:
[[8 3 5 2 7]
 [6 3 6 2 3]
 [7 7 3 7 8]
 [4 7 8 5 6]
 [5 7 7 7 6]]
```

Program 7

Problem Statement : Gene Expression Algorithm for Resource Allocation in Business.

Algorithm :

Algorithm

i> Define the problem.

ii> Initialize parameters => Set population size, mutation rate, crossover rate and no. of generation.

iii> Initialize population => Generate random allocations as "chromosome".

iv> Evaluate fitness => Calculate the fitness score of each chromosome based on objective functions.

Date / /

v> Selection => Select the best chromosomes (solutions based on fitness scores.

vi> Crossover => Combine parts of two parent chromosomes to produce offspring.

vii> Mutation => Introduce small changes in the offspring to explore new solutions.

viii> Gene expression => Convert the genetic representation of the resource allocation into a

ix> Iterate => Repeat the selection, crossover, mutation and evaluation steps for several generations

x> Output the best solution with the highest fitness.

Code:

```
import random

# Define the problem
NUM_PROJECTS = 5  # Number of tasks/projects
RESOURCES = 100   # Total resources (e.g., budget or hours)
POPULATION_SIZE = 20
MUTATION_RATE = 0.1
GENERATIONS = 50

# Fitness function: Profit is proportional to the resources allocated
def fitness_function(allocation, profits):
    total_profit = sum(allocation[i] * profits[i] for i in range(NUM_PROJECTS))
    return total_profit

# Generate an initial random chromosome (resource allocation)
def generate_chromosome():
    allocation = [random.randint(0, RESOURCES) for _ in range(NUM_PROJECTS)]
    total = sum(allocation)
    # Normalize to ensure total resources = RESOURCES
    return [int((res / total) * RESOURCES) for res in allocation]

# Initialize population
def initialize_population():
    return [generate_chromosome() for _ in range(POPULATION_SIZE)]

# Selection: Select the top 50% of the population based on fitness
def selection(population, profits):
    population = sorted(population, key=lambda x: fitness_function(x, profits), reverse=True)
    return population[:POPULATION_SIZE // 2]

# Crossover: Combine two parents to create a child
def crossover(parent1, parent2):
    point = random.randint(1, NUM_PROJECTS - 1)
    child = parent1[:point] + parent2[point:]
    # Normalize to meet resource constraint
    total = sum(child)
    return [int((res / total) * RESOURCES) for res in child]

# Mutation: Randomly adjust a resource allocation
def mutate(chromosome):
    if random.random() < MUTATION_RATE:
        index = random.randint(0, NUM_PROJECTS - 1)
        change = random.randint(-5, 5)
        chromosome[index] = max(0, chromosome[index] + change)
    total = sum(chromosome)
    return [int((res / total) * RESOURCES) for res in chromosome]

# Main GEA algorithm
def gene_expression_algorithm():
    # Random profit coefficients for projects
```

```
    profits = [random.randint(1, 10) for _ in range(NUM_PROJECTS)]
    print("Profit per resource for each project:", profits)

    # Step 1: Initialize population
    population = initialize_population()

    # Step 2: Iterate for generations
    for generation in range(GENERATIONS):
        # Evaluate and select
        selected = selection(population, profits)

        # Create next generation
        new_population = selected.copy()
        while len(new_population) < POPULATION_SIZE:
            parent1, parent2 = random.choice(selected), random.choice(selected)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

        population = new_population

        # Print the best solution in this generation
        best_solution = max(population, key=lambda x: fitness_function(x, profits))
        print(f"Generation {generation + 1}, Best Profit: {fitness_function(best_solution, profits)}")

    # Output the best solution
    best_solution = max(population, key=lambda x: fitness_function(x, profits))
    print("\nBest Resource Allocation:", best_solution)
    print("Maximum Profit Achieved:", fitness_function(best_solution, profits))

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

Output:

```
Profit per resource for each project: [5, 5, 6, 2, 8]
Generation 1, Best Profit: 621
Generation 2, Best Profit: 638
Generation 3, Best Profit: 656
Generation 4, Best Profit: 674
Generation 5, Best Profit: 690
Generation 6, Best Profit: 692
Generation 7, Best Profit: 716
Generation 8, Best Profit: 716
Generation 9, Best Profit: 717
Generation 10, Best Profit: 722
Generation 11, Best Profit: 734
Generation 12, Best Profit: 734
Generation 13, Best Profit: 734
Generation 14, Best Profit: 734
Generation 15, Best Profit: 739
Generation 16, Best Profit: 739
Generation 17, Best Profit: 739
Generation 18, Best Profit: 739
Generation 19, Best Profit: 739
Generation 20, Best Profit: 739
Generation 21, Best Profit: 743
Generation 22, Best Profit: 743
Generation 23, Best Profit: 743
Generation 24, Best Profit: 743
Generation 25, Best Profit: 743
Generation 26, Best Profit: 743
Generation 27, Best Profit: 743
Generation 28, Best Profit: 743
Generation 29, Best Profit: 743
Generation 30, Best Profit: 743
Generation 31, Best Profit: 743
Generation 32, Best Profit: 743
Generation 33, Best Profit: 743
Generation 34, Best Profit: 748
Generation 35, Best Profit: 748
Generation 36, Best Profit: 748
Generation 37, Best Profit: 748
Generation 38, Best Profit: 748
Generation 39, Best Profit: 748
Generation 40, Best Profit: 748
Generation 41, Best Profit: 748
Generation 42, Best Profit: 748
Generation 43, Best Profit: 748
Generation 44, Best Profit: 748
Generation 45, Best Profit: 748
Generation 46, Best Profit: 748
Generation 47, Best Profit: 748
Generation 48, Best Profit: 748
Generation 49, Best Profit: 748
Generation 50, Best Profit: 748

Best Resource Allocation: [1, 1, 23, 0, 75]
Maximum Profit Achieved: 748
```