VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

TANMAY BHARADWAJ (1BM22CS303)

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 **Tanmay Bharadwaj (1BM22CS303),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Prof. Megha J Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/TanmayBj23/BIS_LAB

Program 1:

Genetic Algorithm for Optimization Problems:

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm/Pseudo Code:

```
→ Python Code Description for Optimization of Travelling Salesman
Problem using Genetic Algorithm.

Function Genetic Algorithm (Population size, generations, distance matrix)
    Initialize population with parameters such as
    random tours (permutation of cities)

    For generation from 1 to end Do
        Calculate fitness for each tour in population
        fitness = 1 / (total distance of tour)

    Create empty list next-generation.
    while next-generation is not full do
        parent1 = Selection (population, fitness-score)
        parent2 = Selection (population, fitness-score)
        child = crossover (parent1, parent2).
        child = mutate (child)
        Add child to next-generation.

best-tour = Find the tour with the best fitness in populatn
best-distance = 1/ best-fitness
Return best-tour, best-distance
End Function.

Function Crossover (parent1, parent2)
    Select two crossover points
    Create child by combing segments from parent 1 & parent2
    return child
End Function
```

Function Mutate (tour):
    If random_value < mutation_rate then
        Select two positions in the tour
        Swap the cities at those positions
    End if
    Return Tour
End function

Function Selection (population, fitness_scores)
    Select a candidate based on fitness scores
    return selected candidate
End function

3/10

Code:

```
#Travelling Salesman Problem Application
import random
import numpy as np

# Step 1: Define the distance matrix for 10 cities
def generate_distance_matrix(num_cities=10):
    """Generates a random distance matrix for num_cities."""
    matrix = np.random.randint(1, 100, size=(num_cities, num_cities))
    np.fill_diagonal(matrix, 0)  # Distance from a city to itself is 0
    return matrix

# Fitness function: Total distance of the tour
def fitness_function(individual, distance_matrix):
    """Calculate the total distance of the path described by the individual."""
```

```
    total_distance = sum(distance_matrix[individual[i]][individual[i + 1]] for i in range(len(individual)
- 1))
    total_distance += distance_matrix[individual[-1]][individual[0]]  # Return to start
    return 1 / total_distance  # Inverse of distance, because lower distance = higher fitness

# Step 2: Initialize population (a list of random city tours)
def create_initial_population(population_size, num_cities):
    """Creates an initial population of random tours."""
    return [list(np.random.permutation(num_cities)) for _ in range(population_size)]

# Step 3: Selection using tournament selection
def tournament_selection(population, fitness_values, k=3):
    """Selects the best individual from a random sample of the population."""
    selected = random.sample(list(zip(population, fitness_values)), k)
    return max(selected, key=lambda x: x[1])[0]

# Step 4: Crossover using partially matched crossover (PMX)
def pmx_crossover(parent1, parent2):
    """Performs Partially Matched Crossover (PMX) on two parents."""
    size = len(parent1)
    child1, child2 = [-1] * size, [-1] * size
    p1, p2 = sorted(random.sample(range(size), 2))  # Random crossover points
    child1[p1:p2], child2[p1:p2] = parent1[p1:p2], parent2[p1:p2]

    def fill_child(child, parent):
        for i in range(size):
            if child[i] == -1:
                for gene in parent:
                    if gene not in child:
                        child[i] = gene
                        break
    fill_child(child1, parent2)
    fill_child(child2, parent1)
    return child1, child2

# Step 5: Mutation by swapping two cities in the tour
def mutate(individual, mutation_rate=0.01):
    """Swaps two cities in the tour with a given mutation rate."""
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(individual)), 2)
        individual[i], individual[j] = individual[j], individual[i]  # Swap cities
    return individual

# Step 6: Genetic Algorithm Main Loop
def genetic_algorithm_tsp(distance_matrix, population_size=50, generations=10,
mutation_rate=0.01):
    num_cities = len(distance_matrix)
    population = create_initial_population(population_size, num_cities)

    for generation in range(generations):
```

```
        # Step 7: Evaluate fitness of the population
        fitness_values = [fitness_function(individual, distance_matrix) for individual in population]

        # Track the best individual in this generation
        best_individual = population[fitness_values.index(max(fitness_values))]
        best_fitness = max(fitness_values)
        print(f"Generation {generation+1}: Best fitness = {best_fitness:.4f}, Best individual =
{best_individual}")

        # Step 8: Create a new population
        new_population = []
        while len(new_population) < population_size:
            # Step 9: Selection
            parent1 = tournament_selection(population, fitness_values)
            parent2 = tournament_selection(population, fitness_values)

            # Step 10: Crossover
            child1, child2 = pmx_crossover(parent1, parent2)

            # Step 11: Mutation
            new_population.append(mutate(child1, mutation_rate))
            if len(new_population) < population_size:
                new_population.append(mutate(child2, mutation_rate))

        population = new_population[:population_size]

    # Step 12: Output the best solution found
    fitness_values = [fitness_function(individual, distance_matrix) for individual in population]
    best_individual = population[fitness_values.index(max(fitness_values))]
    best_fitness = max(fitness_values)
    return best_individual, 1 / best_fitness  # Return the best tour and its total distance

# Running the Genetic Algorithm on a randomly generated TSP problem
distance_matrix = generate_distance_matrix(10)  # 10 cities
best_tour, best_distance = genetic_algorithm_tsp(distance_matrix, generations=10)  # 10 generations
print(f"Best tour found: {best_tour}")
print(f"Total distance of the best tour: {best_distance:.2f}")
```

Output:

```
Generation 1: Best fitness = 0.0038, Best individual = [2, 7, 6, 9, 1, 3, 0, 4, 5, 8]
Generation 2: Best fitness = 0.0043, Best individual = [2, 7, 6, 3, 1, 8, 9, 0, 4, 5]
Generation 3: Best fitness = 0.0040, Best individual = [2, 7, 5, 3, 1, 8, 9, 0, 4, 6]
Generation 4: Best fitness = 0.0041, Best individual = [2, 7, 6, 4, 5, 1, 3, 9, 0, 8]
Generation 5: Best fitness = 0.0041, Best individual = [2, 6, 7, 5, 8, 1, 3, 9, 0, 4]
Generation 6: Best fitness = 0.0044, Best individual = [6, 2, 9, 5, 1, 3, 8, 4, 0, 7]
Generation 7: Best fitness = 0.0048, Best individual = [2, 6, 7, 5, 1, 3, 8, 9, 0, 4]
Generation 8: Best fitness = 0.0048, Best individual = [2, 6, 7, 5, 1, 3, 8, 9, 0, 4]
Generation 9: Best fitness = 0.0048, Best individual = [6, 2, 7, 5, 1, 3, 8, 9, 0, 4]
Generation 10: Best fitness = 0.0048, Best individual = [2, 6, 7, 5, 1, 3, 8, 9, 0, 4]
Best tour found: [5, 2, 7, 6, 1, 3, 8, 9, 0, 4]
Total distance of the best tour: 205.00
```

Program 2:

Particle Swarm Optimization for Function Optimization:

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm/Pseudo Code:

→ Fitness function used : Rastrigin function.

$$f(x_1 \dots x_n) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)$$

minimum value at $f(0,0)$ is 0.

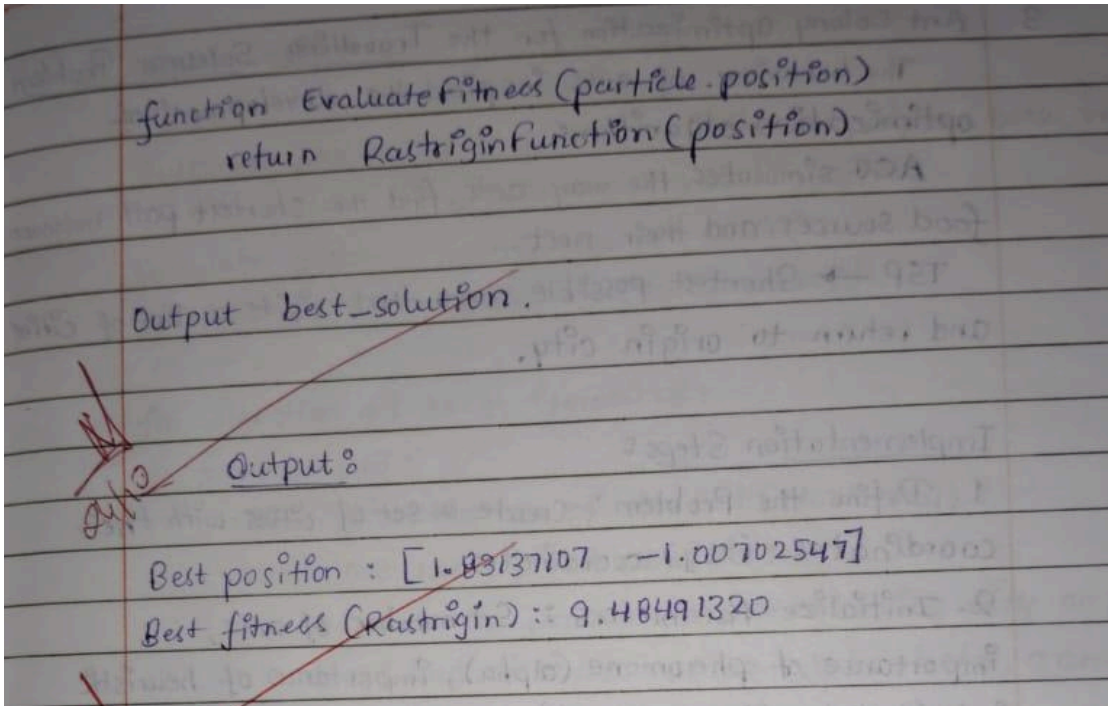
Pseudo Code:

function PSO (num_particles, bounds, num_iterations)
    Initialise parameters such as inertia_weight, coefficients.
    Initialise particles as empty list.

    for i from 1 to num_particles,
        create a particle within the bound range.
        add each particle to the particle list.

best_position = particles[0].best_position
best_fitness is calculated by evaluation function by pushing its best position.

```
from 1 to num_iterations:
    for each particle in particles:
        fitness = EvaluateFitness(particle.position)
        if fitness < particle.best_fitness
            particle.best_fitness = fitness
            particle.best_position = particle.position

    if fitness < best_fitness:
        best_fitness = fitness
        best_position = particle.position.

    r1, r2 = Randomvalues().
    particle.velocity = ( inertia_weight * particle.velocity
                        + coefficient * r1 * (particle.best_
                        position) + r2 * ( particle.position))

return best_position, best_fittness.

function CreateParticle(bounds):
    position = Random Uniform (bounds)
    velocity = Random Uniform([-1, 1])
    best_position = position.
    best_fitness = EvaluateFitness (position)

return Particle (position, velocity, best_position, best_fitness)
```



function EvaluateFitness(particle.position)
  return RastriginFunction(position)

Output best_solution.

Output:

Best position : [1.83137107 -1.00702547]
Best fitness (Rastrigin) :: 9.48491320

Code:

```
#Rastrigin Function Application
import numpy as np

# Define the Rastrigin function
def rastrigin(x):
   A = 10
   return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i) for x_i in x)

# Particle class to store position, velocity, personal best, and fitness
class Particle:
   def __init__(self, bounds):
      self.position = np.random.uniform(bounds[0], bounds[1], size=len(bounds[0]))
      self.velocity = np.random.uniform(-1, 1, size=len(bounds[0]))
      self.best_position = np.copy(self.position)
      self.best_fitness = rastrigin(self.position)

# Particle Swarm Optimization function
def particle_swarm_optimization(num_particles, bounds, num_iterations):
   # PSO parameters
   inertia_weight = 0.5
   cognitive_coeff = 1.5
   social_coeff = 1.5

   # Initialize particles
   particles = [Particle(bounds) for _ in range(num_particles)]
   global_best_position = particles[0].best_position
   global_best_fitness = particles[0].best_fitness
```

```
    for _ in range(num_iterations):
        for particle in particles:
            # Evaluate fitness
            fitness = rastrigin(particle.position)
            # Update personal best
            if fitness < particle.best_fitness:
                particle.best_fitness = fitness
                particle.best_position = np.copy(particle.position)

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

            # Update velocity and position
            r1, r2 = np.random.rand(), np.random.rand()
            particle.velocity = (inertia_weight * particle.velocity +
                                 cognitive_coeff * r1 * (particle.best_position - particle.position) +
                                 social_coeff * r2 * (global_best_position - particle.position))
            particle.position += particle.velocity

        return global_best_position, global_best_fitness

# Parameters
bounds = (np.array([-5, -5]), np.array([5, 5]))  # Bounds for x and y
num_particles = 30
num_iterations = 100

# Run PSO
best_position, best_fitness = particle_swarm_optimization(num_particles, bounds, num_iterations)

print(f"Best Position: {best_position}")
print(f"Best Fitness (Rastrigin): {best_fitness}")
```

Output:

```
Best Position: [ 1.83137107 -1.00702547]
Best Fitness (Rastrigin): 9.484913204947988
```

Program 3:

Ant Colony Optimization for the Traveling Salesman Problem:

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city**.**

Algorithm / PseudoCode:

PseudoCode

Initialise parameters : n_ants, n_iterations, alpha, beta, rho, Q (pheromone value), cities.

Calculate distance matrix for all cities

Initialise pheromone matrix (small values)

for iteration = 1 to n_iterations:
    for each ant:
        Initialise the tour of the ant (start city).
        While ant not visited all cities:
            Calculate the probability next choosing city on pheromone level and distance. using alpha, beta, gamma.
            Update the tour by adding next city.
    Calculate total length for the ant.

    update phermone matrix:
        Evaluate phermone using probabilistic function.
        Deposit phermone on the edges of each ant's tour on the edges it visited.

    Update the best solution found.

    If ~~any~~ Stopping condition is met end for.

Return the best solution found.

Cities = [(0,0), (1,3), (3,1), (6,4), (8,0)]

Code:

```
#Travelling Salesman Problem  Application
import numpy as np
import random
import math

# Function to calculate the distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)

# Function to generate a distance matrix for the cities
def create_distance_matrix(cities):
    n = len(cities)
    distance_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            distance_matrix[i][j] = calculate_distance(cities[i], cities[j])
    return distance_matrix

# Ant Colony Optimization (ACO) Algorithm
def ant_colony_optimization(cities, n_ants, n_iterations, alpha=1, beta=5, rho=0.1, Q=100):
    # Number of cities
    n = len(cities)

    # Initialize pheromone matrix with small values
    pheromone_matrix = np.ones((n, n)) * 1e-6

    # Create distance matrix
    distance_matrix = create_distance_matrix(cities)

    # Best tour found
    best_tour = None
    best_tour_length = float('inf')

    # Main loop
    for iteration in range(n_iterations):
        # Initialize ants
        all_ants_tours = []
        all_ants_lengths = []

        # Loop through each ant
        for ant in range(n_ants):
            tour = []
            visited = [False] * n
            current_city = random.randint(0, n - 1)  # Start at a random city
            visited[current_city] = True
            tour.append(current_city)

            # Construct the solution (tour) for each ant
```

```
            for _ in range(n - 1):
                # Calculate probabilities of visiting each city
                probabilities = []
                for city in range(n):
                    if not visited[city]:
                        pheromone = pheromone_matrix[current_city][city] ** alpha
                        heuristic = (1.0 / distance_matrix[current_city][city]) ** beta
                        probabilities.append(pheromone * heuristic)
                    else:
                        probabilities.append(0)

                # Normalize the probabilities
                total = sum(probabilities)
                probabilities = [prob / total for prob in probabilities]

                # Choose the next city based on the probabilities (roulette wheel selection)
                next_city = np.random.choice(range(n), p=probabilities)
                tour.append(next_city)
                visited[next_city] = True
                current_city = next_city

            # Calculate the length of the tour
            tour_length = 0
            for i in range(n):
                tour_length += distance_matrix[tour[i]][tour[(i + 1) % n]]

            # Update the best tour if found a shorter one
            if tour_length < best_tour_length:
                best_tour_length = tour_length
                best_tour = tour

            # Store the ant's tour and its length
            all_ants_tours.append(tour)
            all_ants_lengths.append(tour_length)

        # Update pheromone matrix
        pheromone_matrix *= (1 - rho)  # Evaporate pheromone

        # Deposit pheromone for each ant's tour
        for ant in range(n_ants):
            pheromone_deposit = Q / all_ants_lengths[ant]
            for i in range(n):
                pheromone_matrix[all_ants_tours[ant][i]][all_ants_tours[ant][(i + 1) % n]] +=
pheromone_deposit

        print(f"Iteration {iteration + 1}/{n_iterations}, Best Length: {best_tour_length}")

    return best_tour, best_tour_length

# Example usage
```

```
if __name__ == "__main__":
    # Define the cities (x, y coordinates)
    cities = [
        (0, 0),  # City 0
        (1, 3),  # City 1
        (3, 1),  # City 2
        (6, 4),  # City 3
        (8, 0),  # City 4
    ]

    # Parameters
    n_ants = 10
    n_iterations = 20

    # Run the ACO algorithm
    best_tour, best_tour_length = ant_colony_optimization(cities, n_ants, n_iterations)

    # Output the best tour and its length
    print(f"Best Tour: {best_tour}")
    print(f"Best Tour Length: {best_tour_length}")
```

Output:

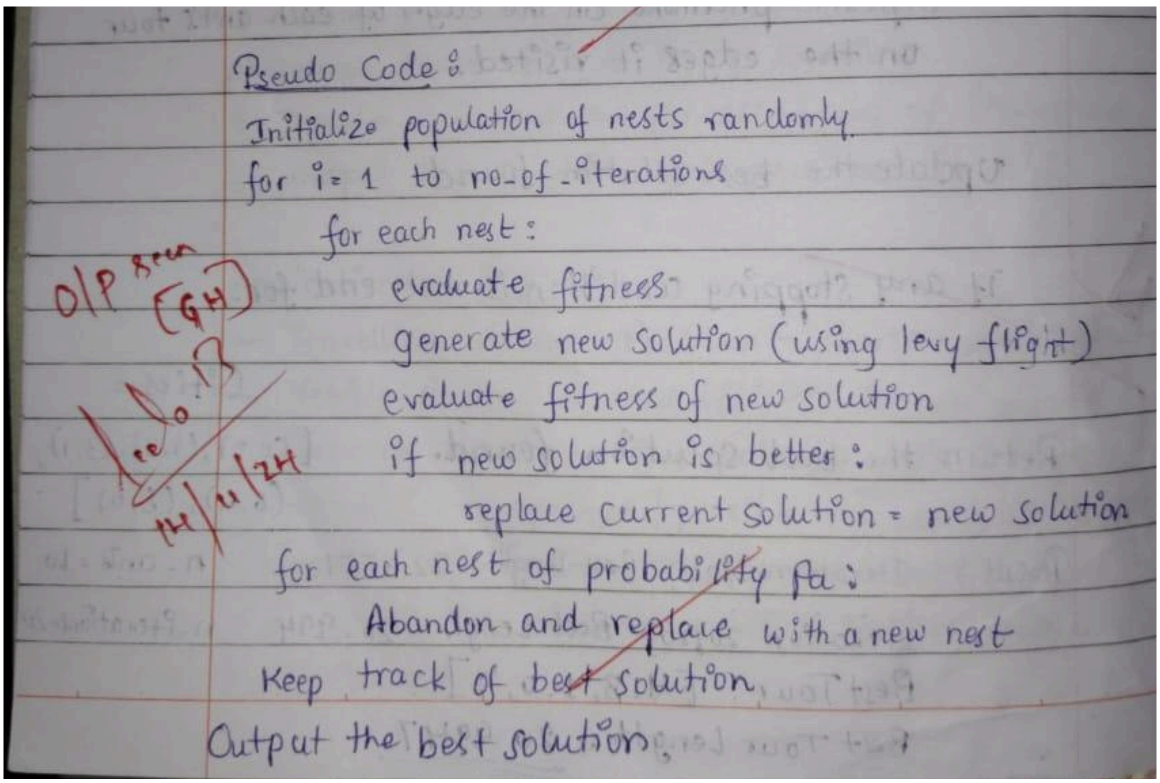
```
Iteration 1/20, Best Length: 22.655105068319532
Iteration 2/20, Best Length: 20.99473030252191
Iteration 3/20, Best Length: 20.99473030252191
Iteration 4/20, Best Length: 20.99473030252191
Iteration 5/20, Best Length: 20.99473030252191
Iteration 6/20, Best Length: 20.99473030252191
Iteration 7/20, Best Length: 20.99473030252191
Iteration 8/20, Best Length: 20.99473030252191
Iteration 9/20, Best Length: 20.99473030252191
Iteration 10/20, Best Length: 20.99473030252191
Iteration 11/20, Best Length: 20.99473030252191
Iteration 12/20, Best Length: 20.99473030252191
Iteration 13/20, Best Length: 20.99473030252191
Iteration 14/20, Best Length: 20.99473030252191
Iteration 15/20, Best Length: 20.99473030252191
Iteration 16/20, Best Length: 20.99473030252191
Iteration 17/20, Best Length: 20.99473030252191
Iteration 18/20, Best Length: 20.99473030252191
Iteration 19/20, Best Length: 20.99473030252191
Iteration 20/20, Best Length: 20.99473030252191
Best Tour: [4, 3, 1, 0, 2]
Best Tour Length: 20.99473030252191
```

Program 4:

Cuckoo Search (CS):

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm/Pseudo Code:



Pseudo Code:
Initialize population of nests randomly
for i = 1 to no-of-iterations
    for each nest:
        evaluate fitness
        generate new solution (using levy flight)
        evaluate fitness of new solution
        if new solution is better:
            replace current solution = new solution
    for each nest of probability pa:
        Abandon and replace with a new nest
    Keep track of best solution
Output the best solution

O/P seen (GH)
14/11/24

Code:

```
#Engineering Design Problem Application
import numpy as np

# Constants
NUM_NESTS = 25  # Number of nests
NUM_ITERATIONS = 10  # Number of iterations
PA = 0.25  # Probability of abandoning a nest
L = 100  # Length of the truss (dummy value for simplicity)
RHO = 7.85e3  # Density of steel (kg/m^3) for the material
```

```
# Function to calculate the weight of the truss structure
def calculate_weight(areas, lengths, rho=RHO):
    return np.sum(areas * lengths * rho)

# Function to calculate the deflection (simplified model for demonstration purposes)
def calculate_deflection(areas, lengths):
    # Example: Deflection = Sum of inverse areas (simplified for the demonstration)
    return np.sum(1 / areas)

# Objective function: Minimize weight while satisfying deflection constraint
def objective_function(areas, lengths, deflection_limit=10):
    weight = calculate_weight(areas, lengths)
    deflection = calculate_deflection(areas, lengths)
    if deflection > deflection_limit:
        return np.inf  # Return an infeasible solution if deflection exceeds the limit
    return weight

# Generate a random initial solution (cross-sectional areas)
def initialize_nests(num_nests, num_beams):
    return np.random.uniform(low=0.1, high=10.0, size=(num_nests, num_beams))

# Cuckoo Search Algorithm
def cuckoo_search(num_nests, num_iterations, areas_range=(0.1, 10.0), lengths=None):
    num_beams = len(lengths) if lengths is not None else 5  # Default: 5 beams
    nests = initialize_nests(num_nests, num_beams)  # Initialize nests with random solutions
    fitness = np.array([objective_function(nest, lengths) for nest in nests])  # Evaluate initial solutions
    best_nest = nests[np.argmin(fitness)]  # Best nest with minimum weight
    best_fitness = np.min(fitness)

    for iteration in range(num_iterations):
        # Generate new solutions (cuckoo eggs)
        new_nests = nests + np.random.randn(num_nests, num_beams) * 0.1  # Levy flight step

        # Ensure the solutions are within the allowable range
        new_nests = np.clip(new_nests, areas_range[0], areas_range[1])

        # Evaluate the new nests
        new_fitness = np.array([objective_function(nest, lengths) for nest in new_nests])

        # Abandon nests if necessary (based on probability PA)
        abandon = np.random.rand(num_nests) < PA
        nests[abandon] = new_nests[abandon]
        fitness[abandon] = new_fitness[abandon]

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

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

    return best_nest, best_fitness

# Example usage:
lengths = [5, 5, 5, 5, 5]  # Length of each beam (in meters)
best_design, best_weight = cuckoo_search(NUM_NESTS, NUM_ITERATIONS, lengths=lengths)

print("Best Design (Cross-sectional areas):", best_design)
print("Best Weight:", best_weight)
```

Output:

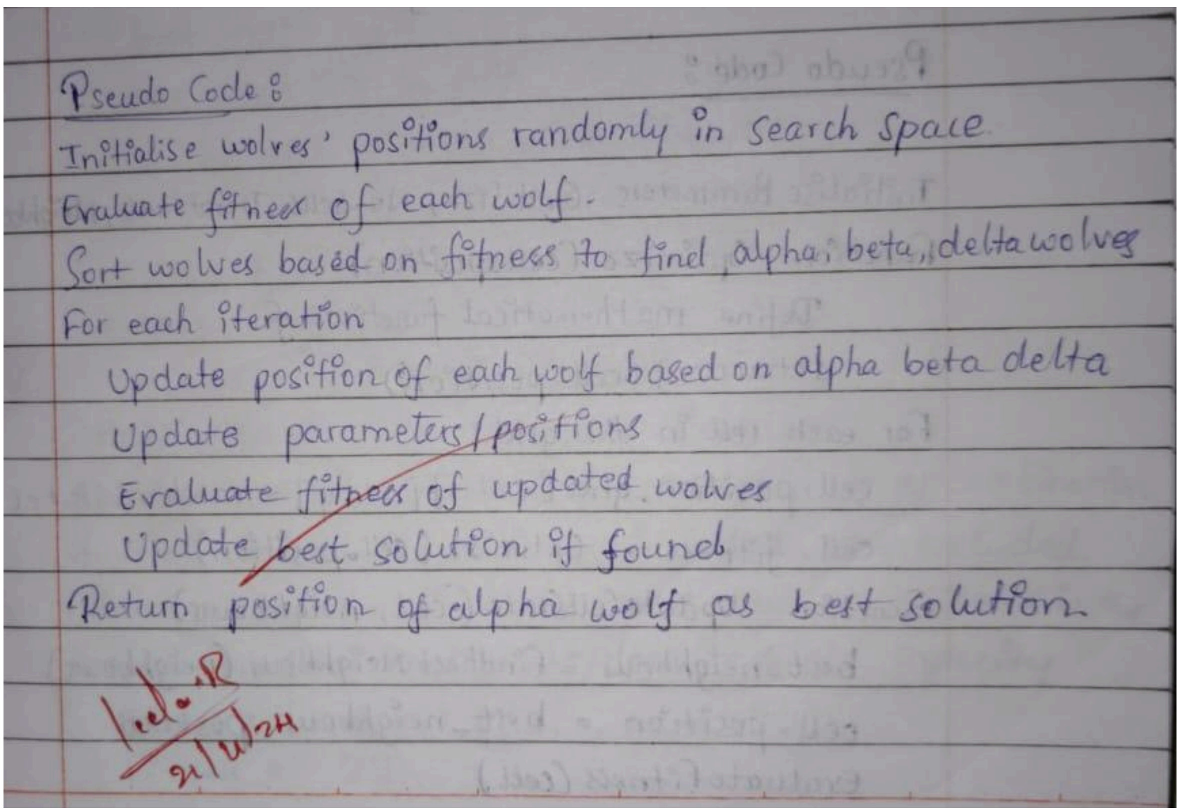
```
Iteration 1/10: Best Fitness = 355357.8430367811
Iteration 2/10: Best Fitness = 355357.8430367811
Iteration 3/10: Best Fitness = 355357.8430367811
Iteration 4/10: Best Fitness = 355357.8430367811
Iteration 5/10: Best Fitness = 355357.8430367811
Iteration 6/10: Best Fitness = 352428.8428649454
Iteration 7/10: Best Fitness = 352428.8428649454
Iteration 8/10: Best Fitness = 352428.8428649454
Iteration 9/10: Best Fitness = 352428.8428649454
Iteration 10/10: Best Fitness = 352428.8428649454
Best Design (Cross-sectional areas): [0.30382941 1.76710058 0.80922782 0.70894598 5.5213804 ]
Best Weight: 352428.8428649454
```

Program 5:

Grey Wolf Optimizer (GWO):

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

Algorithm/Pseudo Code:



Pseudo Code :
Initialise wolves' positions randomly in search space
Evaluate fitness of each wolf.
Sort wolves based on fitness to find alpha, beta, delta wolves
For each iteration
Update position of each wolf based on alpha beta delta
Update parameters/positions
Evaluate fitness of updated wolves
Update best solution if found
Return position of alpha wolf as best solution.

Code:

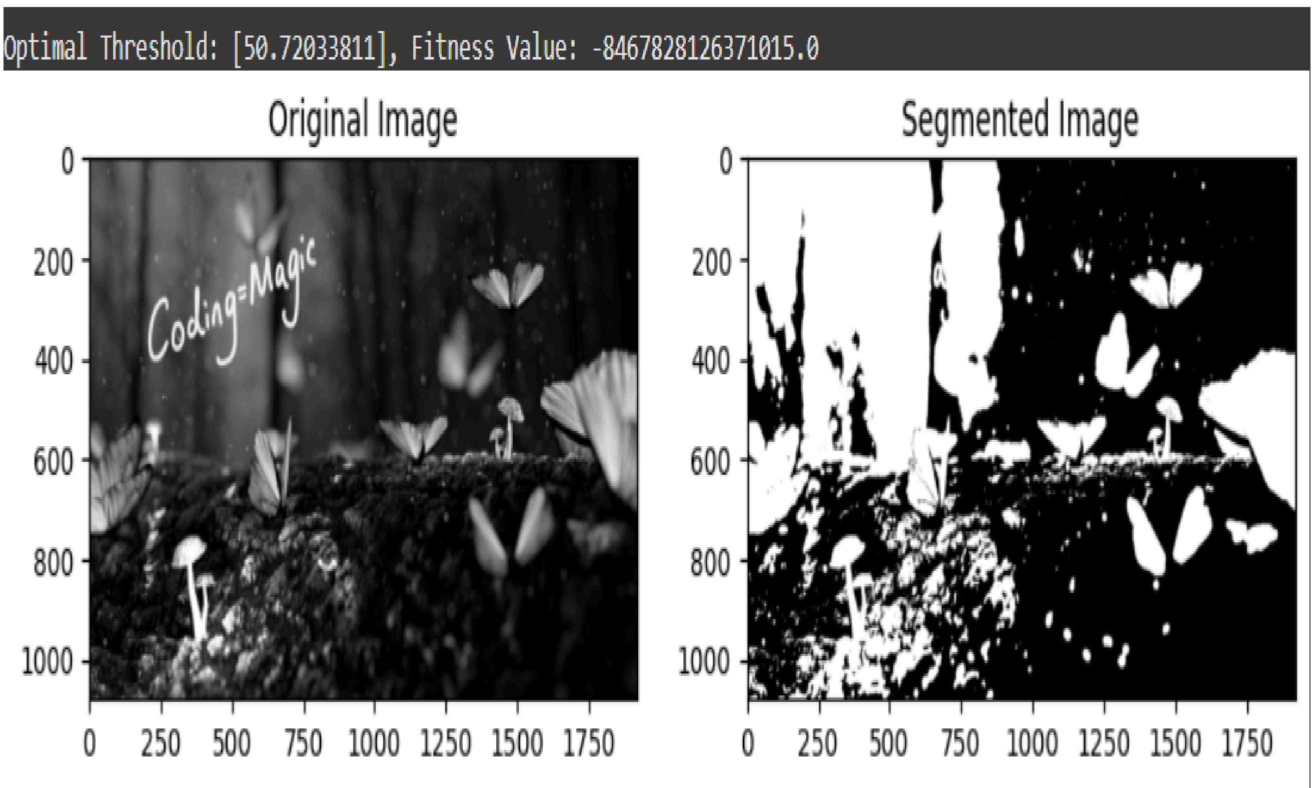
```
#Image Processing Application
import numpy as np
import cv2
import matplotlib.pyplot as plt

class GreyWolfOptimizer:
    def __init__(self, fitness_func, dim, lb, ub, population_size=30, max_iter=50):
        self.fitness_func = fitness_func
        self.dim = dim
        self.lb = lb
        self.ub = ub
```

```
        self.population_size = population_size
        self.max_iter = max_iter

    def optimize(self):
        # Initialize positions of wolves
        wolves = np.random.uniform(self.lb, self.ub, (self.population_size, self.dim))
        fitness = np.array([self.fitness_func(w) for w in wolves])

        # Identify alpha, beta, and delta
        alpha = wolves[np.argmin(fitness)]
        beta = wolves[np.argsort(fitness)[1]]
        delta = wolves[np.argsort(fitness)[2]]
        alpha_score, beta_score, delta_score = np.min(fitness), fitness[np.argsort(fitness)[1]],
fitness[np.argsort(fitness)[2]]

        # Optimization loop
        for t in range(self.max_iter):
            a = 2 - t * (2 / self.max_iter)  # Decreasing linear component

            for i in range(self.population_size):
                for j in range(self.dim):
                    # Update wolves' positions
                    r1, r2 = np.random.rand(), np.random.rand()
                    A1, C1 = 2 * a * r1 - a, 2 * r2
                    D_alpha = abs(C1 * alpha[j] - wolves[i, j])
                    X1 = alpha[j] - A1 * D_alpha

                    r1, r2 = np.random.rand(), np.random.rand()
                    A2, C2 = 2 * a * r1 - a, 2 * r2
                    D_beta = abs(C2 * beta[j] - wolves[i, j])
                    X2 = beta[j] - A2 * D_beta

                    r1, r2 = np.random.rand(), np.random.rand()
                    A3, C3 = 2 * a * r1 - a, 2 * r2
                    D_delta = abs(C3 * delta[j] - wolves[i, j])
                    X3 = delta[j] - A3 * D_delta

                    # Calculate new position
                    wolves[i, j] = np.clip((X1 + X2 + X3) / 3, self.lb[j], self.ub[j])

            # Evaluate fitness
            fitness = np.array([self.fitness_func(w) for w in wolves])

            # Update alpha, beta, delta
            alpha = wolves[np.argmin(fitness)]
            beta = wolves[np.argsort(fitness)[1]]
            delta = wolves[np.argsort(fitness)[2]]
            alpha_score = np.min(fitness)

        return alpha, alpha_score
```

```
# Image Processing Application: Image Thresholding
def image_thresholding_fitness(thresholds, image):
    """Fitness function: Maximizing Otsu's between-class variance."""
    thresholds = thresholds.astype(int)
    thresholds = np.clip(thresholds, 0, 255)
    hist = cv2.calcHist([image], [0], None, [256], [0, 256]).flatten()
    total = hist.sum()
    weight_b, mean_b, sum_b = 0, 0, 0
    max_var = 0

    for i in range(256):
        weight_b += hist[i]
        weight_f = total - weight_b
        if weight_b == 0 or weight_f == 0:
            continue

        sum_b += i * hist[i]
        mean_b = sum_b / weight_b
        mean_f = (hist[i:].dot(np.arange(i, 256))) / weight_f

        # Calculate between-class variance
        between_class_variance = weight_b * weight_f * (mean_b - mean_f) ** 2
        if between_class_variance > max_var:
            max_var = between_class_variance

    return -max_var  # Minimize negative of the variance


if __name__ == "__main__":
    # Load a grayscale image
    image = cv2.imread('sample_image.jpg', 0)  # Provide your grayscale image path here

    # Set parameters for GWO
    gwo = GreyWolfOptimizer(
        fitness_func=lambda x: image_thresholding_fitness(x, image),
        dim=1,
        lb=[0],
        ub=[255],
        population_size=30,
        max_iter=50,
    )

    # Run GWO to find the optimal threshold
    optimal_threshold, fitness_value = gwo.optimize()
    print(f"Optimal Threshold: {optimal_threshold}, Fitness Value: {fitness_value}")

    # Apply thresholding to the image
    _, segmented_image = cv2.threshold(image, int(optimal_threshold[0]), 255,
```

```
cv2.THRESH_BINARY)

    # Display the original and segmented images
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(image, cmap='gray')
    plt.subplot(1, 2, 2)
    plt.title("Segmented Image")
    plt.imshow(segmented_image, cmap='gray')
    plt.show()
```

Output:

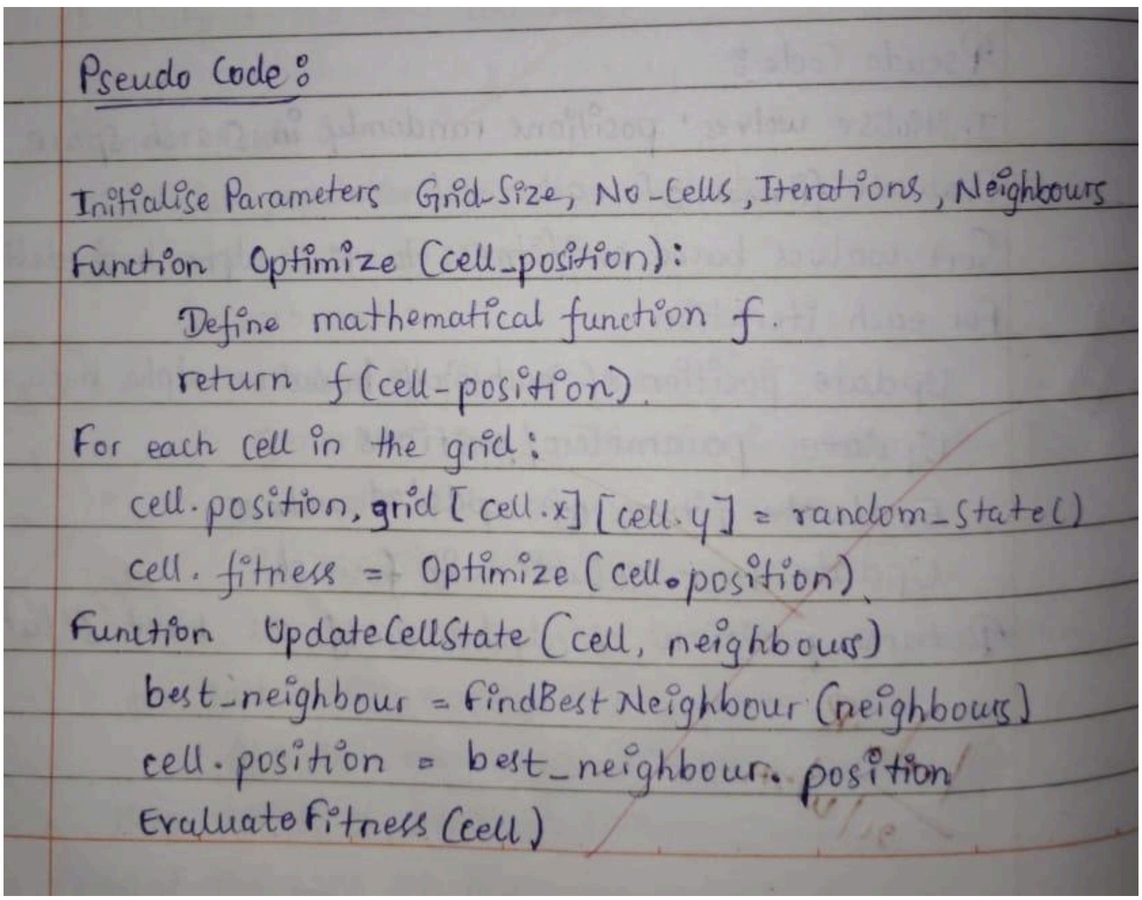


Program 6:

Parallel Cellular Algorithms and Programs:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm/Pseudo Code:



Pseudo Code:

```
Initialise Parameters  Grid-Size, No-Cells, Iterations, Neighbours
Function Optimize (cell_position):
    Define mathematical function f.
    return f(cell-position).
For each cell in the grid:
  cell.position, grid [cell.x] [cell.y] = random_state()
  cell.fitness = Optimize (cell.position)
Function UpdateCellState (cell, neighbours)
  best_neighbour = findBest Neighbour (neighbours)
  cell.position = best_neighbour.position
  EvaluateFitness (cell)
```

```
Function FindBestNeighbour(neighbours):
    best_fitness = Infinity
    best-neighbour = null
    for each neighbour in neighbours :
        If neighbour.fitness < best_fitness:
            best_fitness = neighbour.fitness
            best-neighbour = neighbour.
    return best_neighbour.

For iteration = 1 to Iterations
    Parallel for each cell in the grid :
        neighbour = GetNeighours(cell, neighbours)
        Update CellState(cell, neighbours)
    best-cell = FindBestSolution(grid)

Function FindBestSolution(grid):
    best_fitness = Infinity
    best-cell = null
    For each cell in grid
        If cell-fitness < bestfitness :
            best_fitness = cell-fitness
            best-cell = cell
    return best_cell.
```

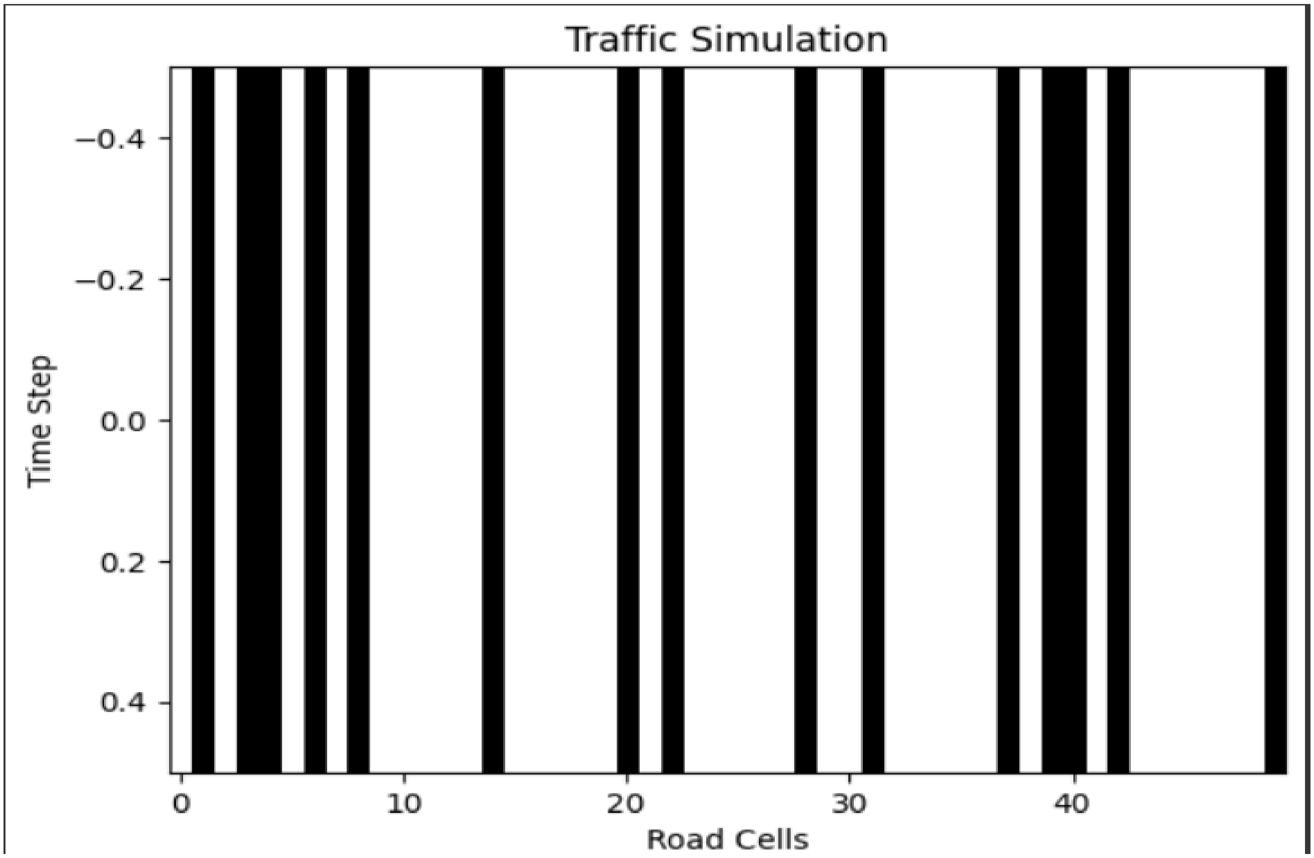
"Traffic Simulation."

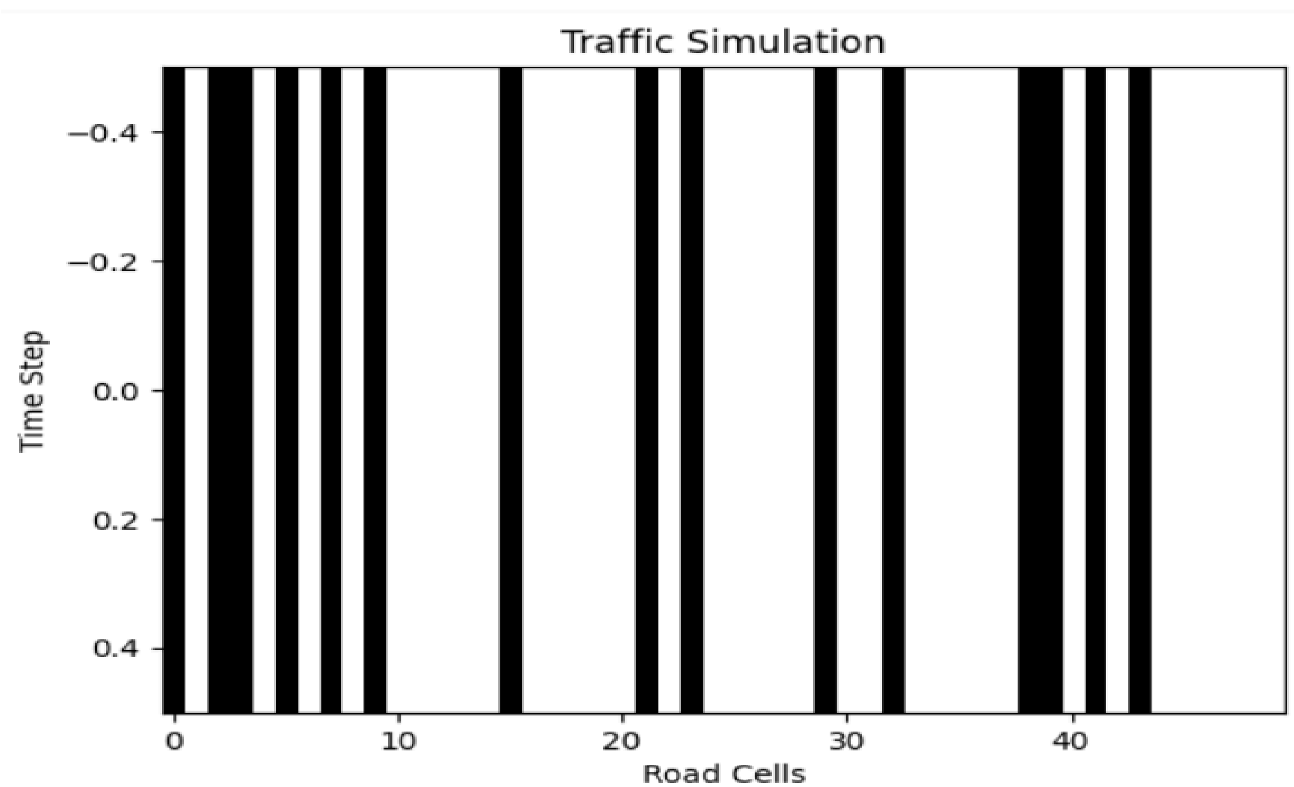
Code:

```
#Traffic Simulation Problem Application
import numpy as np
import matplotlib.pyplot as plt
import time

def initialize_road(length, car_density):
    road = np.zeros(length, dtype=int)
    num_cars = int(car_density * length)
    car_positions = np.random.choice(length, num_cars, replace=False)
    road[car_positions] = 1
    return road
```

```
def update_road(road):
    new_road = road.copy()
    for i in range(len(road)):
        if road[i] == 1 and road[(i + 1) % len(road)] == 0:  # Check if the next cell is empty
            new_road[i] = 0
            new_road[(i + 1) % len(road)] = 1
    return new_road

def simulate_traffic(length, car_density, steps, display=True):
    road = initialize_road(length, car_density)
    history = [road.copy()]

    for _ in range(steps):
        road = update_road(road)
        history.append(road.copy())

        if display:
            plt.imshow([road], cmap="binary", aspect="auto")
            plt.title("Traffic Simulation")
            plt.xlabel("Road Cells")
            plt.ylabel("Time Step")
            plt.pause(0.1)

    if display:
        plt.show()

    return history

if __name__ == "__main__":
    # Simulation parameters
    road_length = 50       # Number of cells
    car_density = 0.3      # Fraction of road occupied by cars
    time_steps = 2         # Number of time steps to simulate

    # Run the simulation
    traffic_history = simulate_traffic(road_length, car_density, time_steps)
```

Output:


Traffic Simulation
−0.4
−0.2
0.0
0.2
0.4
Time Step
0
10
20
30
40
Road Cells


Traffic Simulation
−0.4
−0.2
0.0
0.2
0.4
Time Step
0
10
20
30
40
Road Cells

<u>Program 7:</u>

Optimization via Gene Expression Algorithms:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm/Pseudo Code:

Pseudo Code:

Initialize Parameters Population_Size, Genes, Generations, CrossOver Rate, MutationRate
Define fitness function
Initialize ~~Population~~ each gene in random population
Evaluate fitness function for each in the population
Function Selection (population, fitness_value)
    if fitness_value(for i) > fitness_value(j)
    i. append (select)

```
            (genes)
while next-generation not null
    parent1 = Selection (population, fitnessvalue)
    parent2 = Selection(population, fitness_value)
    child = Crossover (parent1, parent2)
    child = Mutate (child)
    Add child to next genes

Output best-solution (max (fitness_value) ) population
Best_value = max (fitness_values).
```

Code:

```
import numpy as np
import random

def fitness_function(x):
    return np.sin(x) * x

POPULATION_SIZE = 20
NUM_GENES = 1  # Each genetic sequence represents a single variable (x)
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
NUM_GENERATIONS = 20
DOMAIN = (-10, 10)  # Search space for x


def initialize_population():
    return [np.random.uniform(DOMAIN[0], DOMAIN[1], NUM_GENES) for _ in
range(POPULATION_SIZE)]

# Evaluate the fitness of each genetic sequence
def evaluate_fitness(population):
    return [fitness_function(individual[0]) for individual in population]

# Select genetic sequences based on their fitness (roulette wheel selection)
def select_population(population, fitness):
    fitness_sum = sum(fitness)
```

```
    probabilities = [f / fitness_sum for f in fitness]
    selected = random.choices(population, probabilities, k=POPULATION_SIZE)
    return selected

# Perform crossover between selected sequences to produce offspring
def crossover(parent1, parent2):
    if NUM_GENES == 1 or random.random() >= CROSSOVER_RATE:
        # Return parents directly if crossover is not applicable or skipped
        return parent1, parent2
    # Perform crossover at a random point
    point = random.randint(1, NUM_GENES - 1) if NUM_GENES > 1 else 0
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
    return child1, child2

# Apply mutation to the offspring to introduce variability
def mutate(individual):
    for i in range(NUM_GENES):
        if random.random() < MUTATION_RATE:
            individual[i] = np.random.uniform(DOMAIN[0], DOMAIN[1])
    return individual

# Gene expression: translate genetic sequences into functional solutions
# (Already represented directly by the genetic sequence)

def gene_expression_algorithm():
    # Step 1: Initialize Population
    population = initialize_population()

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

        # Track the best solution in the population
        best_fitness = max(fitness)
        best_individual = population[fitness.index(best_fitness)]

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

        # Step 3: Selection
        selected_population = select_population(population, fitness)

        # Step 4: Crossover
        next_population = []
        for i in range(0, POPULATION_SIZE, 2):
            parent1 = selected_population[i]
            parent2 = selected_population[(i + 1) % POPULATION_SIZE]
            child1, child2 = crossover(parent1, parent2)
            next_population.append(child1)
            next_population.append(child2)
```

```
        # Step 5: Mutation
        population = [mutate(individual) for individual in next_population]

    # Output the best solution
    fitness = evaluate_fitness(population)
    best_fitness = max(fitness)
    best_individual = population[fitness.index(best_fitness)]
    print(f"Best Solution: x = {best_individual[0]}, Fitness = {best_fitness}")

# Run the Gene Expression Algorithm
gene_expression_algorithm()
```

Output:

```
Generation 1: Best Fitness = 7.753999060432997
Generation 2: Best Fitness = 7.753999060432997
Generation 3: Best Fitness = 7.837525088480672
Generation 4: Best Fitness = 4.781706816876702
Generation 5: Best Fitness = -2.534055213190368
Generation 6: Best Fitness = 1.5421049089659034
Generation 7: Best Fitness = 1.9527263537460373
Generation 8: Best Fitness = 2.7396722976765546
Generation 9: Best Fitness = -0.9023514966096238
Generation 10: Best Fitness = 0.40915237089145545
Generation 11: Best Fitness = 1.8170045915270605
Generation 12: Best Fitness = 0.3226293955709163
Generation 13: Best Fitness = -2.193991290735217
Generation 14: Best Fitness = 0.004294760782383895
Generation 15: Best Fitness = -2.5698553731614746
Generation 16: Best Fitness = 1.7346788741650052
Generation 17: Best Fitness = 3.822582515872763
Generation 18: Best Fitness = 0.8659809696271437
Generation 19: Best Fitness = 3.676324863242478
Generation 20: Best Fitness = -0.3073177871589717
Best Solution: x = 8.853399909459359, Fitness = 4.787845353561675
```