VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Varun Vinod (1BM22CS322)

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 **Varun Vinod (1BM22CS322),** 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.

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

Index

Github Link: https://github.com/Vatrun/BIS-Lab

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:

Date 3/10/24

Page

LAB 1 - GENETIC ALGORITHMS

-> Used to solve optimization problems in machine learning

-> Phases :

-> Initialization :

Population -> chromosomes -> Genes

-> Chromosomes are initialized using random binary strings

-> Fitness Assignment.

- Fitness function determines how fit an individual is and gives a score to each individual.

-> ~~Algorithm~~. 2 operations

1) Crossover : crossover point at random -> swaps info. Parent genes also exchanged

2) Mutation -> insert random genes in offspring. Solves premature ~~or~~ convergency

-> Algorithm.

1) Start

2) Create initial population

3) Calculate fitness score for each individual

4) repeat : selection

crossover

mutation

calculate fitness score.

until convergence found.

5) Choose individual with highest fitness score

6) Stop

Date __/__/__

Page ______

→ ~~Repetition~~ ~~to return best chromosome from~~

→ Applications.

1) Optimization Problems: GA's are widely used to find optimal solutions in complex space such as: Scheduling, Resource allocation, Route planning, machine learning

2) Engineering Design: Used for optimising design parameters in fields like structural engineering, aerospace, automation

3) Financial Modeling: GA's help in optimizing investment portfolio & ~~profit~~ market trends

4) Game Development: Used to evolve strat for non-player characters

5) Robotics,

6) Artificial Intelligence

7) Art & music.

Code:


```
import random
import numpy as np
import math

def generate_cities(num_cities):
    return [(random.randint(0, 100), random.randint(0, 100)) for _ in range(num_cities)]

def compute_distance_matrix(cities):
    num_cities = len(cities)
    distances = [[0] * num_cities for _ in range(num_cities)]
    for i in range(num_cities):
        for j in range(num_cities):
            if i != j:
                distances[i][j] = math.sqrt(
                    (cities[i][0] - cities[j][0])**2 + (cities[i][1] - cities[j][1])**2
                )
    return distances

class TSP:
    def __init__(self, distances):
        self.distances = distances
        self.num_cities = len(distances)

    def fitness(self, route):

        total_distance = sum(
            self.distances[route[i]][route[i + 1]] for i in range(len(route) - 1)
        )
        total_distance += self.distances[route[-1]][route[0]]

class GeneticAlgorithm:
    def __init__(self, tsp, population_size=100, generations=500, mutation_rate=0.1):
        self.tsp = tsp
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.population = self._initialize_population()

    def _initialize_population(self):
        return [random.sample(range(self.tsp.num_cities), self.tsp.num_cities) for _ in range(self.population_

    def _select_parents(self):

        fitnesses = [self.tsp.fitness(route) for route in self.population]
        total_fitness = sum(fitnesses)
        probabilities = [f / total_fitness for f in fitnesses]
        return random.choices(self.population, probabilities, k=2)

    def _crossover(self, parent1, parent2):

        size = len(parent1)
        start, end = sorted(random.sample(range(size), 2))
        child = [-1] * size
        child[start:end] = parent1[start:end]

        p2_idx = 0
        for i in range(size):
            if child[i] == -1:
                while parent2[p2_idx] in child:
                    p2_idx += 1
                child[i] = parent2[p2_idx]
        return child
```

```
    def _mutate(self, route):

        if random.random() < self.mutation_rate:
            i, j = random.sample(range(len(route)), 2)
            route[i], route[j] = route[j], route[i]

    def evolve(self):
        for _ in range(self.generations):
            new_population = []
            for _ in range(self.population_size):
                parent1, parent2 = self._select_parents()
                child = self._crossover(parent1, parent2)
                self._mutate(child)
                new_population.append(child)
            self.population = new_population

    def get_best_solution(self):

        best_route = min(self.population, key=lambda route: 1 / self.tsp.fitness(route))
        best_distance = 1 / self.tsp.fitness(best_route)
        return best_route, best_distance

if __name__ == "__main__":
    num_cities = 5
    cities = generate_cities(num_cities)
    distances = compute_distance_matrix(cities)

    print("City Coordinates:")
    for i, city in enumerate(cities):
        print(f"City {i}: {city}")

    tsp = TSP(distances)
    ga = GeneticAlgorithm(tsp, population_size=50, generations=100, mutation_rate=0.2)
    ga.evolve()
    best_route, best_distance = ga.get_best_solution()

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

```
City Coordinates:
City 0: (22, 31)
City 1: (33, 46)
City 2: (89, 0)
City 3: (65, 0)
City 4: (3, 17)

Best route: [2, 1, 0, 4, 3]
Best distance: 202.96101904990562
```

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:

Date 24/10/24

Page

LAB : 2

Particle swarm Optimization (PSO)

→ ~~Used to observe~~

PSO is a versatile optimization techinque used in many fields

The algorithm progresses through few phases:-

1) Initialization : Generates a swarm of particles with random positions and velocities
   - Define hyperparameters

2) Evaluation:
   Calculate Fitness of each particle based on the objective function.

3) Updation:
   - Veclocity update : update current velocity, best known position of particle & swarm.
   - Position update
   - Clipping.

4) Termination
   Check for convergence criteria and terminate if met. otherwise return to evaluation phase.

Date __/__/__
Page ____

Algorithm

Step1: Randomly initalize swarm population of
n particles x_i $(i=1,2,...n)$

Step2: Select hyperparam values
$\omega, c1, c2$.

Step3: ~~For iter in range (max_iter)~~:
~~for i in~~ n
For each particle i, update velocity
$v_i = \omega \cdot v_i + r_1 \cdot c_1 (best\_pos - x_i) + r_2 \cdot c_2 \cdot (best\_pos - x_i)$

b) update position
$x_i += v_i$

c) clip position
$x_i = max(min(x_i, maxx), minx)$

d) update Personal & Global Bests
- calculate Fitness
- if $Fitness_i < bestFitness_i$ update bestFitness & bestpos.
- if $Fitness_i <$ best fitness. swarm, update best_fitness_swarm and best_pos_swarm

4) Return:
- Output the best position best_pos_swarm and corresponding fitness best_fitness_swarm

24/10

O/P Best position [2.128313 -07 -1.383564 -07]
Best Fitness: 6.4434712 e-14

Code:


```
import random
import numpy as np

def objective_function(position):
    """The function to be minimized."""
    x, y = position
    return x**2 + y**2

def pso(objective_function, dimensions, iterations, population_size, w=0.7, c1=1.4, c2=1.4):
    """
    Particle Swarm Optimization algorithm.

    Args:
        objective_function: The function to be minimized.
        dimensions: The number of dimensions of the search space.
        iterations: The number of iterations to run the algorithm.
        population_size: The number of particles in the swarm.
        w: Inertia weight.
        c1: Cognitive parameter.
        c2: Social parameter.

    Returns:
        A tuple containing the best solution found and its corresponding objective function value.
    """
    particles = []
    for _ in range(population_size):
        position = np.random.uniform(-10, 10, dimensions)
        velocity = np.random.uniform(-1, 1, dimensions)
        particles.append({
            'position': position,
            'velocity': velocity,
            'best_position': position.copy(),
            'best_value': objective_function(position)
        })

    global_best_position = particles[0]['best_position'].copy()
    global_best_value = particles[0]['best_value']

    for _ in range(iterations):
        for particle in particles:

            r1 = random.random()
            r2 = random.random()
            particle['velocity'] = (w * particle['velocity'] +
                                    c1 * r1 * (particle['best_position'] - particle['position']) +
                                    c2 * r2 * (global_best_position - particle['position']))

            particle['position'] = particle['position'] + particle['velocity']

            particle['position'] = np.clip(particle['position'], -10, 10)

            value = objective_function(particle['position'])

            if value < particle['best_value']:
                particle['best_value'] = value
                particle['best_position'] = particle['position'].copy()

            if value < global_best_value:
                global_best_value = value
                global_best_position = particle['position'].copy()

    return global_best_position, global_best_value
```



```
dimensions = 2
iterations = 100
population_size = 50

best_position, best_value = pso(objective_function, dimensions, iterations, population_size)
print(f"Best position found: {best_position}")
print(f"Best value found: {best_value}")
```

```
Best position found: [ 4.04789703e-08 -2.23363404e-08]
Best value found: 2.137459138638845e-15
```

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:

Date ___ Page ___

7/11/24

Ant Colony Optimization

Algorithm

1. Initialization :
   - Generate random cities in a 2D space
   - Distance matrix to represent distance b/w each pair of cities. Symmetric matrix
   - Pheromone matrix is initialized with pheromone levels set to 1 for all pairs.

2. ACO loop:
   - In each iteration,
     1) Ants construct a tour by probabilistically moving from city to city based on ~~[illegible]~~ pheromone levels. Ants choose cities with higher pheromone levels.
     2) ~~Aft~~ All ants have constructed the tour and best one is selected
     3) Pheromone matrix gets updated based on tours found by ants. Pheromone evaporate over time.

3) Pheromone Updation
   1) All pheromone levels are reduced by (1-RHO) to simulate evaporation.
   2) Pheromone levels are increased on the edges visited by the ants, where amount pheromone is inversely proportional to tour length

Date __/__/__
Page ______

- Result: Algorithm stops after a predefined no of iterations
- The best tour is plotted on 2D graph.

Seela.R
8/10/24

Code:


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

# Define constants for the algorithm
NUM_ANTS = 50
NUM_CITIES = 20  # Now we have 20 cities
ALPHA = 1.0  # Influence of pheromone
BETA = 2.0   # Influence of distance
RHO = 0.1    # Pheromone evaporation rate
Q = 100       # Pheromone deposit constant
MAX_ITER = 100  # Maximum number of iterations

# Predefined 20 cities (coordinates in 2D space)
def generate_cities():
    cities = np.array([
        [5, 10], [11, 5], [14, 9], [12, 15], [8, 13],  # Cities 0-4
        [10, 10], [13, 7], [16, 5], [14, 3], [18, 6],  # Cities 5-9
        [4, 2], [7, 1], [8, 5], [6, 7], [4, 10],  # Cities 10-14
        [15, 18], [12, 17], [3, 18], [17, 12], [19, 8]  # Cities 15-19
    ])
    return cities

# Compute the distance matrix
def compute_distance_matrix(cities):
    num_cities = len(cities)
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            dist = np.linalg.norm(cities[i] - cities[j])
            distance_matrix[i, j] = dist
            distance_matrix[j, i] = dist
    return distance_matrix

# Initialize pheromone matrix
def initialize_pheromone_matrix(num_cities):
    pheromone_matrix = np.ones((num_cities, num_cities))  # Pheromone starts as 1 for all edges
    np.fill_diagonal(pheromone_matrix, 0)  # No pheromone on the diagonal (self-loops)
    return pheromone_matrix

# Calculate the total length of a tour
def calculate_tour_length(tour, dist_matrix):
    length = 0
    for i in range(len(tour) - 1):
        length += dist_matrix[tour[i], tour[i + 1]]
    length += dist_matrix[tour[-1], tour[0]]  # Returning to the start
    return length

# Ant solution construction (probabilistic decision on next city)
def construct_solution(num_cities, pheromone_matrix, dist_matrix):
    tour = [random.randint(0, num_cities - 1)]  # Start from a random city
    visited = set(tour)

    while len(tour) < num_cities:
        current_city = tour[-1]
        probabilities = []
        for next_city in range(num_cities):
            if next_city not in visited:
                pheromone = pheromone_matrix[current_city, next_city] ** ALPHA
                distance = (1.0 / dist_matrix[current_city, next_city]) ** BETA
                probabilities.append(pheromone * distance)
            else:
                probabilities.append(0)
```

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

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

        return tour

    # Update the pheromone matrix based on the solutions found by ants
    def update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour):
        # Evaporate pheromone
        pheromone_matrix *= (1 - RHO)

        # Add pheromone for all ants
        for tour in all_tours:
            tour_length = calculate_tour_length(tour, dist_matrix)
            for i in range(len(tour) - 1):
                pheromone_matrix[tour[i], tour[i + 1]] += Q / tour_length
            pheromone_matrix[tour[-1], tour[0]] += Q / calculate_tour_length(tour, dist_matrix)

        # Add pheromone for the best tour
        best_length = calculate_tour_length(best_tour, dist_matrix)
        for i in range(len(best_tour) - 1):
            pheromone_matrix[best_tour[i], best_tour[i + 1]] += Q / best_length
        pheromone_matrix[best_tour[-1], best_tour[0]] += Q / best_length

    # Main ACO algorithm for solving TSP
    def ant_colony_optimization(num_cities, dist_matrix, pheromone_matrix, max_iter):
        best_tour = None
        best_tour_length = float('inf')

        # Main loop
        for iteration in range(max_iter):
            all_tours = []

            # Step 1: All ants construct their solutions
            for _ in range(NUM_ANTS):
                tour = construct_solution(num_cities, pheromone_matrix, dist_matrix)
                all_tours.append(tour)
                tour_length = calculate_tour_length(tour, dist_matrix)

                # Step 2: Update the best tour if necessary
                if tour_length < best_tour_length:
                    best_tour = tour
                    best_tour_length = tour_length

            # Step 3: Update pheromone matrix
            update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour)

            # Optional: print progress every 10 iterations
            if iteration % 10 == 0:
                print(f"Iteration {iteration}: Best tour length = {best_tour_length:.2f}")

        return best_tour, best_tour_length


    # Main Execution
    if __name__ == "__main__":
        # Step 1: Generate predefined cities and distance matrix
        cities = generate_cities()
        dist_matrix = compute_distance_matrix(cities)
```

```
# Step 2: Initialize pheromone matrix
pheromone_matrix = initialize_pheromone_matrix(NUM_CITIES)

# Step 3: Run ACO algorithm
best_tour, best_tour_length = ant_colony_optimization(NUM_CITIES, dist_matrix, pheromone_matrix, MAX_ITER)

# Step 4: Output the best tour and visualize it
print(f"Best tour length: {best_tour_length:.2f}")
```

```
Iteration 0: Best tour length = 107.48
Iteration 10: Best tour length = 81.48
Iteration 20: Best tour length = 80.59
Iteration 30: Best tour length = 80.50
Iteration 40: Best tour length = 79.23
Iteration 50: Best tour length = 79.23
Iteration 60: Best tour length = 77.88
Iteration 70: Best tour length = 77.88
Iteration 80: Best tour length = 77.88
Iteration 90: Best tour length = 77.88
Best tour length: 77.88
```

Program 4

Problem statement: Implement the Cuckoo Search Algorithm for feature selection to identify an optimal subset of features that maximizes the classification accuracy of a Support Vector Machine (SVM) on a given dataset.

Algorithm:

Date
Page

11/11/24

LAB 04
Cuckoo Search Algorithm

Algorithm
Initialize: Population of n cuckoos
Evaluate fitness of each cuckoo
#input
n = initial population size
Pa = fraction of worse nests to be abandoned and replaced
Max_iterations : Max no of iterations
f = the objective fn to optimize()

Generate the initial population of n nest
x_i (i=1, 2 -- n)
While t < Max iterations :
Get a cuckoo randomly by levy flight
Evaluate its quality fitness F_i

j ∈ choose a nest among n randomly
if $F_i > F_j$
Replace j by the new solutions

if A fraction (Pa) of worse nests are abandoned & new ones are built :
keep the best solution
Rank the solutions & find the current best
Post process result

Application of vehicle routing problem
It is a classic optimization problem in logistics & transportation where the goal to determin the optimal set of routes

Date __/__/__
Page ____

for fleet of vehicles to deliver the goods. The objective is to minimize the total distance travelled or overall cost

Parameters
N -> population size
max_iter - No of max iterations
Pa -> Abandon Probability
α - Controls step size distribution
μ - is characteristic exponent of levy flight distribution

O/P
Best solution (route):
Route: [3, 9, 2]
Route: [6, 1, 8]
Route: [5, 7, 4]

Best total distance: 8284.86776...

Code:


```
#cuckoo search(Traffic Signal Optimization)
import numpy as np
from scipy.special import gamma

def fitness_function(x):
    waiting_times = np.array([10 + (x[i] ** 2) / 100 for i in range(len(x))])
    total_waiting_time = np.sum(waiting_times)
    return total_waiting_time

def levy_flight(dim, beta=1.5):
    sigma_u = np.power((gamma(1 + beta) * np.sin(np.pi * beta / 2) /
                        gamma((1 + beta) / 2) * beta * (2 ** (beta - 1))), 1 / beta)
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, 1, dim)
    step = u / np.power(np.abs(v), 1 / beta)
    return step

def cuckoo_search(dim, bounds, num_nests, max_iter, p_a=0.25, Lambda=1.5):
    nests = np.random.uniform(bounds[0], bounds[1], (num_nests, dim))
    fitness = np.array([fitness_function(nest) for nest in nests])

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

    for iter in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(num_nests):
            step = levy_flight(dim, Lambda)
            new_nests[i] = nests[i] + step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])

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

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

        if np.random.rand() < p_a:
            random_idx = np.random.randint(num_nests)
            nests[random_idx] = np.random.uniform(bounds[0], bounds[1], dim)
            fitness[random_idx] = fitness_function(nests[random_idx])

        current_best_idx = np.argmin(fitness)
        current_best_fitness = fitness[current_best_idx]

        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_idx]



    return best_nest, best_fitness

dim = 3
bounds = [10, 120]
num_nests = 20
max_iter = 100

best_solution, best_value = cuckoo_search(dim, bounds, num_nests, max_iter)

print("\n--- Best Solution ---")
```



```
print("Green Light Timings (seconds):", best_solution)
print("Best Fitness Value (Total Waiting Time):", best_value)
```

```
--- Best Solution ---
Green Light Timings (seconds): [10. 10. 10.]
Best Fitness Value (Total Waiting Time): 33.0
```

Program 5

Problem Statement : Implement the Grey Wolf Optimizer (GWO) to optimize the hyperparameters (C and gamma) of a Support Vector Machine (SVM) classifier for achieving the best classification accuracy on the Iris dataset.

Algorithm :

Date __/__/__
Page ____

21/11/24

28/11/2

LAB - 05

Gray Wolf Optimization

Algorithm

Initialize population of wolf with random positions.

Evaluate fitness of each wolf.

While iteration < max_iter:

Sort wolves by fitness

Identify top three wolves (Alpha, beta, γ)

for each wolf:

update position using

Xnew = (alpha, beta, gamma weighted avg based on α β γ)

if new position has better fitness,

update the wolf and update the position and fitness of the wolf

update alpha, beta, and gama wolves with the top 3 wolves

Reduce the value of 'a' (linearly from 2 to 0)

Increment the iteration Counter

Return the position of the alpha wolf or best solution

O/P

Iteration 1 - Best fitness : 45.65710

iteration 2 - Best fitness 26.870057

iteration 3 - Best fitness 26.870057

iteration 4 - Best fitness 26.870057

Best Param : 26.870057

Code:

```
import numpy as np


N_INPUTS = 3
N_HIDDEN = 5
N_OUTPUTS = 1
N_WOLVES = 30
MAX_ITER = 100
LB = -10.0
UB = 10.0

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)


def forward_pass(input_data, weights):
    hidden_layer = np.dot(input_data, weights[:N_INPUTS *
N_HIDDEN].reshape(N_INPUTS, N_HIDDEN))
    hidden_layer = sigmoid(hidden_layer)
    output = np.dot(hidden_layer, weights[N_INPUTS *
N_HIDDEN:].reshape(N_HIDDEN, N_OUTPUTS))
    return output, hidden_layer

def fitness_function(weights, inputs, targets, n_samples):
    total_error = 0.0
    for i in range(n_samples):
        output, _ = forward_pass(inputs[i], weights)
        total_error += (output - targets[i]) ** 2
    return total_error / n_samples


def rand_range(min_val, max_val):
    return min_val + (max_val - min_val) * np.random.random()

def update_position(positions, alpha_pos, beta_pos, delta_pos, i, t):
    A = 2 - t * (2.0 / MAX_ITER)
    C = 2 * np.random.random()

    for j in range(len(positions[i])):

        D_alpha = np.abs(C * alpha_pos[j] - positions[i][j])
        D_beta = np.abs(C * beta_pos[j] - positions[i][j])
        D_delta = np.abs(C * delta_pos[j] - positions[i][j])

        new_position = alpha_pos[j] - A * D_alpha if np.random.random()
> 0.5 else beta_pos[j] - A * D_beta
```

```
            positions[i][j] = np.clip(new_position, LB, UB)


def gwo_optimization(inputs, targets, n_samples):
    positions = np.random.uniform(LB, UB, (N_WOLVES, N_INPUTS *
N_HIDDEN + N_HIDDEN))  # Wolves' positions (weights)
    fitness = np.zeros(N_WOLVES)  # Fitness of wolves
    alpha_pos = np.zeros(N_INPUTS * N_HIDDEN + N_HIDDEN)  # Best
position (alpha wolf)
    beta_pos = np.zeros(N_INPUTS * N_HIDDEN + N_HIDDEN)   # Second best
position (beta wolf)
    delta_pos = np.zeros(N_INPUTS * N_HIDDEN + N_HIDDEN)  # Third best
position (delta wolf)
    alpha_score = float('inf')  # Best fitness value (alpha wolf)
    beta_score = float('inf')   # Second best fitness value (beta wolf)
    delta_score = float('inf')  # Third best fitness value (delta wolf)


    for i in range(N_WOLVES):
        fitness[i] = fitness_function(positions[i], inputs, targets,
n_samples)


        if fitness[i] < alpha_score:
            alpha_score = fitness[i]
            alpha_pos = positions[i]
        elif fitness[i] < beta_score:
            beta_score = fitness[i]
            beta_pos = positions[i]
        elif fitness[i] < delta_score:
            delta_score = fitness[i]
            delta_pos = positions[i]

    for t in range(MAX_ITER):
        for i in range(N_WOLVES):
            update_position(positions, alpha_pos, beta_pos, delta_pos,
i, t)
            fitness[i] = fitness_function(positions[i], inputs,
targets, n_samples)


            if fitness[i] < alpha_score:
                alpha_score = fitness[i]
                alpha_pos = positions[i]
            elif fitness[i] < beta_score:
                beta_score = fitness[i]
                beta_pos = positions[i]
```

```
            elif fitness[i] < delta_score:
                delta_score = fitness[i]
                delta_pos = positions[i]


        if t % 10 == 0:
            print(f"Iteration {t}/{MAX_ITER}, Best Fitness =
{alpha_score}")

    print(f"\nBest Fitness: {alpha_score}")
    return alpha_pos


inputs = np.array([
    [0.0, 0.0, 1.0],
    [1.0, 0.0, 1.0],
    [0.0, 1.0, 1.0],
    [1.0, 1.0, 1.0]
])

targets = np.array([0.0, 1.0, 1.0, 0.0])


best_weights = gwo_optimization(inputs, targets, len(inputs))

print("\nEvaluating the final model with the best weights...")
for i in range(len(inputs)):
    output, _ = forward_pass(inputs[i], best_weights)
    print(f"Input: {inputs[i]}, Predicted Output: {output[0]}, Actual
Target: {targets[i]}")
```

Iteration 0/100, Best Fitness = 0.5000013911617378

Iteration 10/100, Best Fitness = 0.5000013911617378

Iteration 20/100, Best Fitness = 0.5000013911617378

Iteration 30/100, Best Fitness = 0.5000013911617378

Iteration 40/100, Best Fitness = 0.5000013911617378

Iteration 50/100, Best Fitness = 0.5000013911617378

Iteration 60/100, Best Fitness = 0.5000013911617378

Iteration 70/100, Best Fitness = 0.5000013911617378

Iteration 80/100, Best Fitness = 0.5000013911617378

Iteration 90/100, Best Fitness = 0.5000013911617378

Best Fitness: 0.5000013911617378

Evaluating the final model with the best weights...

Input: [0. 0. 1.], Predicted Output: -0.0022698934351217197, Actual Target: 0.0

Input: [1. 0. 1.], Predicted Output: -1.0305768090951018e-07, Actual Target: 1.0

Input: [0. 1. 1.], Predicted Output: -1.0305768090951018e-07, Actual Target: 1.0

Input: [1. 1. 1.], Predicted Output: -4.678811484419649e-12, Actual Target: 0.0

=== Code Execution Successful ===

Program 6

Problem Statement : Develop a parallel cellular automaton-based algorithm for optimal robot route planning in a grid-based environment, ensuring collision-free navigation while minimizing travel distance and computational time.

Algorithm :

Date __/__/__
Page ____

28/11/24

LAB - 06

Parallel Cellular Algorithm.

```
Algorithm.
  #constants
  Grid_width = N
  Grid_height = M
  Max_gen = T
#Initialize
  grid = InitializeGrid(Grid_width, Grid_height)

function Countlive neighbours(grid, i, j):
    live_neighbours = 0
    for each neighbour (x, y) of (i, j):
        if grid[x][y] == 1:
            live_neighbours += 1
    return live_neighbours

function Applyrule(grid, i, j)
    live_neighbours = Countlive neighbours(grid, i, j)
    if grid[i][j] == 1:
        return 1 if live_neighbours == 2 or
                   live_neighbours == 3 else
    else:
        return 1 if live_neighbours == 3 else

Function Update(grid):
    new = copygrid(grid)
    for i in range(0, grid_hight):
        for j in range(0, grid_wid
            new[i][j] = Applyrule(grid, i,
    return new
```

9r

Distance Grid:

[[0 1 2 3 inf
inf inf 3 inf 7
6 5 4 5 6
7 inf inf inf 7
8 9 10 9 8]]

Shortest path:

{(0,0), (0,1), (0,2), (1,2), (2,2), (2,3), (2,4), (3,4), (4,4)}

Code:

```python
import random
import numpy as np

# Initialize Grid (N x M) with random states (0 or 1)
def initialize_grid(N, M):
    return np.random.choice([0, 1], size=(N, M))

# Count live neighbors for a cell (i, j)
def count_live_neighbors(grid, i, j, N, M):
    return sum(
        grid[ni, nj] for ni in range(i-1, i+2) for nj in range(j-1, j+2)
        if 0 <= ni < N and 0 <= nj < M and (ni != i or nj != j)
    )

# Update cell's state based on neighbors' count
def update_cell(grid, new_grid, i, j, N, M):
    live_neighbors = count_live_neighbors(grid, i, j, N, M)
    if grid[i, j] == 1:
        new_grid[i, j] = 1 if live_neighbors in [2, 3] else 0
    else:
        new_grid[i, j] = 1 if live_neighbors == 3 else 0

# Print the current state of the grid (0s and 1s)
def print_grid(grid):
    for row in grid:
        print(' '.join(map(str, row)))
    print()  # For spacing between generations

# Main Game of Life simulation with printing grid
def parallel_game_of_life(N, M, steps):
    grid = initialize_grid(N, M)

    for _ in range(steps):
        print_grid(grid)  # Print current grid state
        new_grid = np.zeros((N, M), dtype=int)  # New grid for next state
        for i in range(N):
            for j in range(M):
                update_cell(grid, new_grid, i, j, N, M)
        grid = new_grid  # Update grid for next iteration

    print_grid(grid)  # Print final grid after all steps

# Example Usage
N, M = 5, 5  # Smaller grid size for readability
steps = 5  # Number of iterations
final_grid = parallel_game_of_life(N, M, steps)
```

Output:

```
0 0 0 0 0
0 1 0 0 0
0 0 0 1 1
0 1 0 1 1
1 1 1 0 1

0 0 0 0 0
0 0 0 0 0
0 0 0 1 1
1 1 0 0 0
1 1 1 0 1

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
1 0 0 0 1
1 0 1 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 1 0 0 0
0 1 0 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

<u>Program 7</u>

Problem Statement : Solve the 0/1 Knapsack Problem using the Gene Expression Algorithm (GEA) to maximize the total value of selected items without exceeding the given weight capacity.

Algorithm :

28/11/24 Gene Expression Algorithm

Date ___/___/___
Page ______

Genetic Expression Algorithm are inspired by the biological process of gene expression in living organism. This process involves the translation of genetic information encoded in DNA into functional proteins. The algorithm evolves their solutions through selection, crossover mutation and gen expression to find optimal or near optimal solutions.

Parameters :
Population
Gene size
Mutation rate
Cross rate
Generations
Fitness Functions

Applications
Engineering optimization
knapsack problem
Data mining
Game Theory

Algorithm :

1. Define the problem
objective function
$f(x) \rightarrow$ maximize / minimize
x is the vector of decision variable

Fitness function :
Fitness $(x) \rightarrow$ evaluate $(f(x))$
Determine how good a solution is

2. Initialize parameters :

Date
Page

N → Population Size
G → Gene Length
M → Mutation Rate
C → Cross Rate
T → No. of generations

3. Initialize population:
for i = 1 to N
individual = Random Solution (G)
population.add (individual)

4. Evaluate fitness
For each individual in population
fitness = evaluateFitness (individual)
fitness values.add (fitness)

5. Selection
6. Crossover
7. Mutation
8. Translate genetic representation in actual sol.
9. Iterate
10. Output the Best Solution.

Output:
Best Solution : 1.9999
Best Fitness : 4.000

18
18/12/24

Code:

```
import random

# Define the Knapsack Problem (Objective Function)
def knapsack_fitness(items, capacity, solution):
    total_weight = sum([items[i][0] for i in range(len(solution)) if solution[i] == 1])
    total_value = sum([items[i][1] for i in range(len(solution)) if solution[i] == 1])

    # If total weight exceeds the capacity, return 0 (invalid solution)
    if total_weight > capacity:
        return 0
    return total_value

# Gene Expression Algorithm (GEA)
class GeneExpressionAlgorithm:
    def __init__(self, population_size, num_items, mutation_rate, crossover_rate, generations,
capacity, items):
        self.population_size = population_size
        self.num_items = num_items
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.capacity = capacity
        self.items = items
        self.population = []

    # Initialize population with random solutions (binary representation)
    def initialize_population(self):
        self.population = [[random.randint(0, 1) for _ in range(self.num_items)] for _ in
range(self.population_size)]

    # Evaluate fitness of the population
    def evaluate_fitness(self):
        return [knapsack_fitness(self.items, self.capacity, individual) for individual in self.population]

    # Select individuals based on fitness (roulette wheel selection)
    def selection(self):
        fitness_values = self.evaluate_fitness()
        total_fitness = sum(fitness_values)
        if total_fitness == 0:  # Avoid division by zero
            return random.choices(self.population, k=self.population_size)
        return random.choices(self.population, weights=[f / total_fitness for f in fitness_values],
k=self.population_size)

    # Crossover (single-point) between two individuals
    def crossover(self, parent1, parent2):
        if random.random() < self.crossover_rate:
            crossover_point = random.randint(1, self.num_items - 1)
```

```
            return parent1[:crossover_point] + parent2[crossover_point:]
        return parent1

    # Mutation (random flip of a gene) of an individual
    def mutation(self, individual):
        if random.random() < self.mutation_rate:
            mutation_point = random.randint(0, self.num_items - 1)
            individual[mutation_point] = 1 - individual[mutation_point]
        return individual

    # Evolve population over generations
    def evolve(self):
        self.initialize_population()
        best_solution = None
        best_fitness = 0

        for gen in range(self.generations):
            # Selection
            selected = self.selection()

            # Crossover and Mutation
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1 = selected[i]
                parent2 = selected[i + 1] if i + 1 < self.population_size else selected[i]

                offspring1 = self.crossover(parent1, parent2)
                offspring2 = self.crossover(parent2, parent1)

                new_population.append(self.mutation(offspring1))
                new_population.append(self.mutation(offspring2))

            self.population = new_population

            # Evaluate fitness and track the best solution
            fitness_values = self.evaluate_fitness()
            max_fitness = max(fitness_values)
            if max_fitness > best_fitness:
                best_fitness = max_fitness
                best_solution = self.population[fitness_values.index(max_fitness)]

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

        return best_solution, best_fitness

# Get user input for the knapsack problem
def get_user_input():
    print("Enter the number of items:")
```

```
    num_items = int(input())
    items = []
    print("Enter the weight and value of each item (space-separated):")
    for i in range(num_items):
        weight, value = map(int, input(f"Item {i + 1}: ").split())
        items.append((weight, value))

    print("Enter the knapsack capacity:")
    capacity = int(input())

    return items, capacity, num_items

# Get user input for GEA parameters
def get_algorithm_parameters():
    print("Enter the population size:")
    population_size = int(input())
    print("Enter the mutation rate (e.g., 0.1 for 10%):")
    mutation_rate = float(input())
    print("Enter the crossover rate (e.g., 0.8 for 80%):")
    crossover_rate = float(input())
    print("Enter the number of generations:")
    generations = int(input())

    return population_size, mutation_rate, crossover_rate, generations

# Main function
if __name__ == "__main__":
    # Get user input
    items, capacity, num_items = get_user_input()
    population_size, mutation_rate, crossover_rate, generations = get_algorithm_parameters()

    # Run GEA
    gea = GeneExpressionAlgorithm(population_size, num_items, mutation_rate, crossover_rate,
generations, capacity, items)
    best_solution, best_fitness = gea.evolve()

    print("\nBest Solution:", best_solution)
    print("Best Fitness (Total Value):", best_fitness)
```

```
Enter the number of items:
5
Enter the weight and value of each item (space-separated):
Item 1: 10 20
Item 2: 30 40
Item 3: 50 60
Item 4: 70 80
Item 5: 90 100
Enter the knapsack capacity:
90
Enter the population size:
100
Enter the mutation rate (e.g., 0.1 for 10%):
0.2
Enter the crossover rate (e.g., 0.8 for 80%):
0.9
Enter the number of generations:
10
Generation 1: Best Fitness = 120
Generation 2: Best Fitness = 120
Generation 3: Best Fitness = 120
Generation 4: Best Fitness = 120
Generation 5: Best Fitness = 120
Generation 6: Best Fitness = 120
Generation 7: Best Fitness = 120
Generation 8: Best Fitness = 120
Generation 9: Best Fitness = 120
Generation 10: Best Fitness = 120

Best solution (items selected): [1, 1, 1, 0, 0]
Best fitness (total value): 120
```