VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shantanu Shrivastav (1BM22CS252)

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 **Shantanu Shrivastav(1BM22CS252),** 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.

Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Shantanuu3/BIS_Lab

Program 1

Genetic Algorithm for Optimization Problems

DATE: 03/08 PAGE:

~~Genetic Problem~~

Genetic Algorithm for Optimization Problems

→ Genetic algorithm are inspired by the process of natural selection and genetics. In GA, a population of potential solution is evolved over generation to find the best solution to a problem. The core idea is to select the fittest individual from the population and use them for reproduction to create the next generation.

GAs are widely used for solving optimization and search problem. In this document, we will explore how to implement a genetic algorithm.

Example: Maximising $f(x) = x2 f(x) = x\hat{}f(x) = x2$

Implementation Steps:

1) Define the problem

The first step is to create a mathematical function that needs to be optimized. This function could represent anything depending on the nature of the problem.

DATE: PAGE:

2) Initialize Parameter
Set the necessary parameters for the GA. These include:
→ population size
→ Mutation Rate
→ Crossover Rate
→ Number of Generation

3) Create initial Problem
Generate an initial ~~problem~~ population of potential solution. This population typically consist of random individuals that represent possible answers.

4) Evaluate fitness
Evaluate the fitness of each individuals in the population. The fitness function is problem-specific and is used to quantify how good a particular solution is

5) Selection
select individual based on their fitness to reproduce and generate offspring for the next generate

6) Crossover
Crossover is the process of combining two parent solution

DATE: PAGE:

to create offspring. It mimics biological reproduction where two parent exchange genetic material to produce a child.

7) Mutation
Mutation introduces genetic diversity by making small changes to an individual' structure.

8) Iteration
Repeat the evaluation, selection, crossover, and mutation process.

Applications in Computer Science and Engineering

1) Machine Learning and Neural Networks
→ ~~hypermarket~~
→ hyperparameter Tuning
→ Feature selection

2) Software Engineering
→ Automated test case gen
→ Refactoring

3) Optimization problem
→ Travelling Salesman
→ Scheduling

4) Robotics :
→ Path Planning
→ Behavioral Optimization

5) Network Design and optimization
→ Routing protocols
→ Network topology design

Conclusion :
→ GA are a versatile and powerful tool for solving complex optimization problems accross many domains in computer science and engineering. Their ability to explore large search spaces.

Complete Ranking Algorithm
3/10

Code:

```
import random

# Set a random seed for reproducibility
random.seed(42)

def fitness(chromosome):
    x = int(''.join(map(str, chromosome)), 2)
    return x ** 2

def binary_string_to_chromosome(binary_string):
    return [int(bit) for bit in binary_string]

def generate_population_from_input():
    population = []
    for _ in range(population_size):
        while True:
            binary_string = input("Enter a binary string of size 5 (e.g., '11001'): ")
            if len(binary_string) == 5 and all(bit in '01' for bit in binary_string):
                population.append(binary_string_to_chromosome(binary_string))
                break
            else:
                print("Invalid input. Please enter a binary string of size 5.")
    return population

def select_pair(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parent1 = population[random.choices(range(len(population)), selection_probs)[0]]
    parent2 = population[random.choices(range(len(population)), selection_probs)[0]]
    return parent1, parent2

def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:point] + parent2[point:]
    offspring2 = parent2[:point] + parent1[point:]
    return offspring1, offspring2

def mutate(chromosome, mutation_rate):
    return [gene if random.random() > mutation_rate else 1 - gene for gene in chromosome]
```

```
# Parameters
population_size = 4
generations = 20
mutation_rate = 0.01

# Initialize population from user input
population = generate_population_from_input()

for generation in range(generations):
    fitnesses = [fitness(chromosome) for chromosome in population]

    new_population = []

    # Create new population
    while len(new_population) < population_size:
        parent1, parent2 = select_pair(population, fitnesses)
        offspring1, offspring2 = crossover(parent1, parent2)
        new_population.append(mutate(offspring1, mutation_rate))
        new_population.append(mutate(offspring2, mutation_rate))

    # Ensure the new population has the right size
    population = new_population[:population_size]

# Get the maximum fitness
fitnesses = [fitness(chromosome) for chromosome in population]
max_fitness = max(fitnesses)

print(f"Maximum Possible Fitness: {max_fitness}")
```

Output:

```
Enter a binary string of size 5 (e.g., '11001'): 11011
Enter a binary string of size 5 (e.g., '11001'): 01011
Enter a binary string of size 5 (e.g., '11001'): 11100
Enter a binary string of size 5 (e.g., '11001'): 01101
Maximum Possible Fitness: 841
```

Program 2

Ant Colony Optimization

DATE: PAGE:

Ant colony Optimization for travelling salesman problem

→ Ant colony optimization is a nature inspired algorithm based on the foraging behaviour of ants, specifically the way ants find the shortest path from their colony to a food source. The algorithm is used for solving combinatorial optimiz problem including TSP.

→ ACO algorithm for TSP

1) Initialization:

- initialize the pheromone levels $\tau_{i,j}$ for all edges b/w cities
- Set the parameters: α, β, number of ants, number of iteration, pheromone

2) Ant's tour Construction:

- For each ant, start from a random city
- At each step, the ant chooses the next city based on the pherome levels and distance.
- Repeat until all cities

DATE: PAGE:

are visited and the tour is completed.

3) Pheromone update

- After all ants have completed their tour, update the pheromones on the edges according to the rule:

$$\tau_{i,j} = (1-\rho)\tau_{ij} + \Delta\tau_{i,j}$$

where:

- ρ is the ~~phenon~~ pheromone evaporation rate.

- $\Delta\tau_{i,j}$ is the amount of pheromone deposited by ants on the edge (i, j) which is inversely prop. to the length

4) Termination

- If the stopping criteria are met stop the algorithm. Otherwise repeat from step 2

DATE: PAGE:

Pseudo Code :

```
initialize pheromone matrix τ[i][j] = τ0
initialize best_tour = empty
initialize best_length = infinity

for iteration = 1 to max_iter
  initialise all ants tours

  for each ant k :
    ant_tour = []
    current_city = random_city()
    ant_tour.append (current_city)

    while tour is not complete
    (all cities visited) :
      next_city = choose_city
                  (current_city,
                   ant_tour)
      ant_tour.append (next_city)
      current_city = next_city

    tour_length = calculate_
                  tour_length
                  (ant_tour)
    if tour_length < best_length :
      best_tour = ant_tour
      best_length = tour_length

  apply pheromone evaporation
  (τ[i][j] = (1-ρ)*τ[i][j])

  for each ant k :
```

DATE: PAGE:

ant_tour = [ ]
current_city = random_city()
ant_tour.append(current_city)

// function to choose the next city
probability = [ ]
total_probability = 0
for each city j not in visited_cities:

pherome = T[current_city][j]^β
total_pheromone += pheromone * visibility

for each city j not in visited_cities:
pheromone = T[current_city][j]^α

visibility = (1/distance[current_city][j])^β
probability = (pheromone * visibility)/total_phenomeo
probabilities.append(probability)
next_city = select_city_based_on_probability(probabilities)
return next_city

ΔT = Q/tour_length
T[i][j] += ΔT

Sneha SB 21/11/24

Code:

```
import random
import numpy as np
import operator

FUNCTIONS = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
TERMINALS = ['x', 1, 2, 3, 4]  # x and constants

def random_gene(length=10):
    return [random.choice(list(FUNCTIONS.keys()) + TERMINALS) for _ in range(length)]

def decode_chromosome(chromosome, x):
    stack = []
    for gene in chromosome:
        if gene in FUNCTIONS:  # If it's a function, pop arguments and apply
            if len(stack) < 2:  # Avoid errors if stack has fewer than 2 elements
                stack.append(0)
                continue
            b = stack.pop()
            a = stack.pop()
            try:
                result = FUNCTIONS[gene](a, b)
            except ZeroDivisionError:
                result = 1  # Avoid division by zero
            stack.append(result)
        elif gene == 'x':
            stack.append(x)
        else:
            stack.append(gene)
    return stack[0] if stack else 0  # Return top of stack as output

def fitness_function(chromosome, target_function, x_values):
    predictions = [decode_chromosome(chromosome, x) for x in x_values]
    targets = [target_function(x) for x in x_values]
    mse = np.mean([(p - t) ** 2 for p, t in zip(predictions, targets)])
    return mse


def selection(population, fitnesses):
    total_fitness = sum(1 / (f + 1e-6) for f in fitnesses)  # Avoid division by zero
```

```
    probabilities = [(1 / (f + 1e-6)) / total_fitness for f in fitnesses]
    return population[np.random.choice(len(population), p=probabilities)]

def mutate(chromosome, mutation_rate=0.1):
    new_chromosome = chromosome[:]
    for i in range(len(new_chromosome)):
        if random.random() < mutation_rate:
            new_chromosome[i] = random.choice(list(FUNCTIONS.keys()) + TERMINALS)
    return new_chromosome

def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def ant_colony_optimization(cost_matrix, n_ants=10, n_iterations=100, evaporation_rate=0.5,
alpha=1, beta=2):
    n_nodes = len(cost_matrix)
    pheromones = np.ones((n_nodes, n_nodes))  # Initialize pheromones

    def calculate_probability(i, j, visited):
        if j in visited:
            return 0
        return (pheromones[i][j] ** alpha) * ((1 / cost_matrix[i][j]) ** beta)

    def construct_solution():
        path = [random.randint(0, n_nodes - 1)]
        while len(path) < n_nodes:
            i = path[-1]
            probabilities = [calculate_probability(i, j, path) for j in range(n_nodes)]
            total = sum(probabilities)
            probabilities = [p / total if total > 0 else 0 for p in probabilities]
            next_node = np.random.choice(range(n_nodes), p=probabilities)
            path.append(next_node)
        path.append(path[0])  # Return to start
        return path

    def path_cost(path):
        return sum(cost_matrix[path[i]][path[i + 1]] for i in range(len(path) - 1))
```

```
    best_path = None
    best_cost = float('inf')

    for iteration in range(n_iterations):
        solutions = [construct_solution() for _ in range(n_ants)]
        costs = [path_cost(solution) for solution in solutions]
        for i, cost in enumerate(costs):
            if cost < best_cost:
                best_cost = cost
                best_path = solutions[i]

        pheromones *= (1 - evaporation_rate)  # Evaporation
        for i, solution in enumerate(solutions):
            for j in range(len(solution) - 1):
                pheromones[solution[j]][solution[j + 1]] += 1 / costs[i]

        print(f"Iteration {iteration + 1}: Best Cost = {best_cost}")

    print("Best Path:", best_path)
    print("Best Cost:", best_cost)

cost_matrix = [
    [0, 2, 2, 5, 7],
    [2, 0, 4, 8, 2],
    [2, 4, 0, 1, 3],
    [5, 8, 1, 0, 2],
    [7, 2, 3, 2, 0]
]
ant_colony_optimization(cost_matrix, n_ants=5, n_iterations=20)
```

Output:

```
Iteration 15: Best Cost = 9
Iteration 16: Best Cost = 9
Iteration 17: Best Cost = 9
Iteration 18: Best Cost = 9
Iteration 19: Best Cost = 9
Iteration 20: Best Cost = 9
Best Path: [1, 0, 2, 3, 4, 1]
Best Cost: 9
```

Program 3

Particle Swarm Optimization

Algorithm:

Particle Swarm Optimization

PSO is a popular optimization algo is inspired by the social behaviour of birds and fish. This particularly effective for optimization especially in complex multidimentional spaces.

PSO Algo :-

1) Initialisation : Set parameters such as no of particles, max iteration personal and social (global) coefficient.

2) Fitness calculation
Evaluate the fitness of each particle using the objective fn

3) updating best positive
Each particle keeps track :
→ personal best position
→ global best position

4) Velocity update : update the velocity of each particle based on its new velocity.

5) Position update : update the position of each particle

DATE: PAGE:

6) Boundary condition: If particles moves out of the defined search space boundaries it can be reset to the boundary

7) Termination condition: The process until a ~~shopping~~ stopping condition:
→ max no. of iteration
→ satisfactory fitness
→ no significant improvement

Pseudo code:
Initialize parameter
N = number of particles
D = dimensionality of the search space
w = inertia weight
c1 = cognitive coefficient
c2 = social coefficient

Initialise each particle:
For each particle i in range (1 to N):
Initialize parameter pos : x[i]
Initialize particle velocity: v(i)

$V[i] = w * \cancel{tt} V[i]$
$+ c[i] * random() * (p\_best - x[i]$

$\cancel{+ c[i]}$
$+ c2 * random * (g\_best -$

DATE: PAGE:

update position
x[i] = x[i] + v[i]

Evaluate fitness
fitness [i] = evaluate (x[i])

update personal best:
if fitness [i] < fitness[p_best]
p_best [i] = x[i]
if fitness [p_best [i]] <
fitness [g_best]:
g[best] = p_best[i]

return g_best as optimal sol

Code:

```
import random
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

def fitness_function(x1, x2):
    f1 = x1 + 2 * -x2 + 3
    f2 = 2 * x1 + x2 - 8
    z = f1**2 + f2**2
    return z

def update_velocity(particle, velocity, pbest, gbest, w_min=0.5, max=1.0, c=0.1):
    new_velocity = np.zeros_like(particle)
    r1 = random.uniform(0, max)
    r2 = random.uniform(0, max)
    w = random.uniform(w_min, max)

    for i in range(len(particle)):
        new_velocity[i] = (w * velocity[i] +
                     c * r1 * (pbest[i] - particle[i]) +
                     c * r2 * (gbest[i] - particle[i]))
    return new_velocity

def update_position(particle, velocity):
    new_particle = particle + velocity
    return new_particle

def pso_2d(population, dimension, position_min, position_max, generation, fitness_criterion):
    # Initialization
    particles = np.array([[random.uniform(position_min, position_max) for _ in range(dimension)] for
_ in range(population)])
    pbest_position = particles.copy()
    pbest_fitness = np.array([fitness_function(p[0], p[1]) for p in particles])

    gbest_index = np.argmin(pbest_fitness)
    gbest_position = pbest_position[gbest_index]

    velocity = np.zeros((population, dimension))
```

```
images = []  # For animation

for t in range(generation):
    if np.average(pbest_fitness) <= fitness_criterion:
        break

    for n in range(population):
        velocity[n] = update_velocity(particles[n], velocity[n], pbest_position[n], gbest_position)
        particles[n] = update_position(particles[n], velocity[n])

    pbest_fitness = np.array([fitness_function(p[0], p[1]) for p in particles])
    for n in range(population):
        if pbest_fitness[n] < fitness_function(pbest_position[n][0], pbest_position[n][1]):
            pbest_position[n] = particles[n]

    gbest_index = np.argmin(pbest_fitness)
    gbest_position = pbest_position[gbest_index]

    # Plotting the current positions of the particles
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')

    x = np.linspace(position_min, position_max, 80)
    y = np.linspace(position_min, position_max, 80)
    X, Y = np.meshgrid(x, y)
    Z = fitness_function(X, Y)
    ax.plot_wireframe(X, Y, Z, color='r', linewidth=0.2)

    ax.scatter3D(
        particles[:, 0],
        particles[:, 1],
        [fitness_function(p[0], p[1]) for p in particles],
        c='b'
    )

    # Capture the frame for animation
    plt.title(f'Generation: {t + 1}')
    plt.tight_layout()
```

```
        plt.savefig(f'frame_{t}.png')
        plt.close(fig)

    # Create animation
    frames = [plt.imread(f'frame_{i}.png') for i in range(t)]
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.axis('off')
    image = ax.imshow(frames[0])

    def update(frame):
        image.set_array(frames[frame])
        return image,

    ani = animation.FuncAnimation(fig, update, frames=len(frames), interval=100)
    ani.save('./pso_simple.gif', writer='pillow')

    # Print the results
    print('Global Best Position: ', gbest_position)
    print('Best Fitness Value: ', min(pbest_fitness))
    print('Average Particle Best Fitness Value: ', np.average(pbest_fitness))
    print('Number of Generations: ', t)

# Run the PSO algorithm
pso_2d(population=30, dimension=2, position_min=-10, position_max=10, generation=100,
fitness_criterion=1e-3)
```

Output:

```
Global Best Position:  [2.59992843 2.79914636]
Best Fitness Value:  3.6691186243893878e-06
Average Particle Best Fitness Value:  0.0007223322365523365
Number of Generations:  45
```

Program 4

Cuckoo Search Algorithm

Algorithm:

DATE: PAGE:

Cuckoo Search Algorithm

Cuckoo Search is an optimization algo that mimics the ~~brood~~ brood parasitism behaviour of cuckoos, leveraging levy flights to explore the complex optimisation problem by mimicking the breeding behaviour of cuckoos and their interaction with host birds.

Algorithm:

① Initialise population of n nests (solution) randomly.

② Evaluate the fitness of all nests.

③ while stopping criterion is not met:

(a) for each nest:

(i) generate a new solution using ~~A~~ levy flights

(ii) evaluate the fitness of new solution

(iii) If the new sol^n is better, replace the old nest with the new one

DATE: PAGE:

(b) Abandon some nests with probability p-a and replace them with new random solⁿ.

④ Return the best solⁿ found

Pseudocode:

① Initialise parameters:
- N: no of nests (popⁿ size)
- Max: Maxⁿ number of iteratⁿ
- Pa: Probability of discovering alien eggs
- Problem: specific bonds for the search space (lower and upper limits)

② Initialise nests (solⁿ)
- For each nest i=1 to N
- Randomly generate initial solⁿ x-i within search space

③ Evaluate fitness of each nest
- for each nest i = 1 to N
- Evaluate fitness (x-i) using the objective functⁿ f(x-i)

④ Set best solⁿ as the initial best solⁿ:
- Set BestSolution = arg min (fitness(x-i))

DATE: PAGE:

(5) for each iteration ($t = 1$ to max):

(a) Generate new sol^n (nests) using Levy flights:

$x\_i' = x\_i + \alpha \cdot$ levy flight

- Ensure x_i' stays within the search space
If x_i is out of bounds adjusts it to stay within the boundaries
- Evaluate fitness (x_i')

(b) Evaluate a new sol^n with the current nests:
- if fitness (x_i') < fitness (x_i):
- Replace the old sol^n x_i with the new sol^n x_i'

(c) Randomly nests with a prob. Pa (~~stso~~ discovery of alien eggs)
- for each nests $i = 1$ to N:
- Evaluate fitness (x - rand)

(d) Update the best sol^n.
- if a better sol^n is found:
- update best solution to the best solution among all nests.

Code:

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

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

# Lévy flight function for generating random steps
def levy_flight(beta=1.5, dim=2):
    sigma_u = np.power(np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) / np.math.gamma((1 +
beta) / 2) / np.power(2, (beta - 1) / 2), 1 / beta)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, sigma_v, dim)
    return u / np.power(np.abs(v), 1 / beta)

# Cuckoo Search Algorithm
class CuckooSearch:
    def __init__(self, func, dim, population_size, max_generations, pa=0.25, beta=1.5, lower_bound=-
5, upper_bound=5):
        self.func = func                # Objective function
        self.dim = dim                  # Dimension of the problem
        self.population_size = population_size  # Number of nests (solutions)
        self.max_generations = max_generations  # Maximum number of generations
        self.pa = pa                    # Probability of alien eggs (nest replacement)
        self.beta = beta                # Lévy flight exponent
        self.lower_bound = lower_bound  # Lower bound of the search space
        self.upper_bound = upper_bound  # Upper bound of the search space

        # Initialize population (nests)
        self.nests = np.random.uniform(self.lower_bound, self.upper_bound, (self.population_size,
self.dim))
        self.fitness = np.array([self.func(nest) for nest in self.nests])  # Fitness of each nest
        self.best_nest = self.nests[np.argmin(self.fitness)]  # Best solution found
        self.best_fitness = np.min(self.fitness)  # Best fitness value

    # Update nests using Lévy flights and objective function evaluations
    def generate_new_nests(self):
```

```
        new_nests = []
        for i in range(self.population_size):
            step = levy_flight(self.beta, self.dim)
            new_nest = self.nests[i] + step
            # Apply boundary check
            new_nest = np.clip(new_nest, self.lower_bound, self.upper_bound)
            new_nests.append(new_nest)
        return np.array(new_nests)

    # Main cuckoo search algorithm
    def search(self):
        history = []  # To record the best fitness values over generations

        for generation in range(self.max_generations):
            # Generate new nests based on Lévy flight
            new_nests = self.generate_new_nests()
            new_fitness = np.array([self.func(nest) for nest in new_nests])

            # Replace nests with new ones if they are better
            for i in range(self.population_size):
                if new_fitness[i] < self.fitness[i] or np.random.rand() < self.pa:
                    self.nests[i] = new_nests[i]
                    self.fitness[i] = new_fitness[i]

            # Find the best nest in the current population
            current_best_fitness = np.min(self.fitness)
            current_best_nest = self.nests[np.argmin(self.fitness)]

            # Update the global best solution
            if current_best_fitness < self.best_fitness:
                self.best_fitness = current_best_fitness
                self.best_nest = current_best_nest

            # Record the best fitness for the current generation
            history.append(self.best_fitness)
            print(f"Generation {generation+1}: Best fitness = {self.best_fitness}")

        return self.best_nest, self.best_fitness, history

# Analyze the Cuckoo Search Algorithm
def analyze_cuckoo_search():
```

```
    # Set up parameters for Cuckoo Search
    dim = 2
    population_size = 50
    max_generations = 100
    cuckoo_search = CuckooSearch(func=rastrigin, dim=dim, population_size=population_size,
max_generations=max_generations)

    # Run the Cuckoo Search algorithm
    best_nest, best_fitness, history = cuckoo_search.search()

    # Plot the convergence curve
    plt.plot(history)
    plt.title("Convergence Curve of Cuckoo Search Algorithm")
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness")
    plt.show()

    print(f"Best solution found: {best_nest}")
    print(f"Best fitness: {best_fitness}")

# Run the analysis
analyze_cuckoo_search()
```

Output:

```
Best solution found: [1.30548027 2.02026344]
Best fitness: 0.16306139523513963
```

Program 5

Grey Wolf Optimizer

Algorithm:

DATE: PAGE:

Grey Wolf Algorithm

- The grey wolf optimization algo is a nature inspired optimization algorithm that mimics the leadership hierarchy and hunting behaviour of grey wolves in the wild. The grey wolf pack has an organized structure with various roles, such as alph (leader), beta, delta and omega. The GWO algorithm leverages these hierarichal ~~relath~~ relationship to guide a populat of candidates.

Algorithm:

1 Initialization - ~~we s~~ we start by randomly initializing the position of the wolves.

2 Fitness evaluation : Each wolf's position is evaluated using a fitness func^n, which determine how good the current solution is.

3 Sorting : The wolves are sorted by their fitness so that the best, second best and third best wolves are identified

4. Position update: in each iteration the position of each wolf is updated by "moving" it closer to the alpha, beta and delta wolves based on their relative position

5 Termination: After a pre defined number of iteration the algo terminates.

→ Pseudocode

1 Initialise the population of grey wolves with random posⁿ

2 Evaluate ~~the~~ fitness of each wolf.

3 Sort the wolves in ascending order based on their fitness

4 Identify the posⁿ of the alpha beta and delta

5 Set the maximum no. of iteration

6 For each iteration (t = 1 to maxIteration):

(a) for each wolf in the posⁿ

DATE: PAGE:

(i) update the position of the wolf based on their pos^n of the alpha, beta, delta

(ii) calculate the new fitness of the wolf.

(iii) if the new fitness is better than previous

b update the pos^n of alpha, beta and delta wolves

(c) Sort the wolves again based on their fitness.

7 Return the pos^n of the alpha wolf as the optimal solution.

Code:

```
import numpy as np

def objective_function(x):
    return np.sum(x**2)

class GreyWolfOptimizer:
    def __init__(self, objective_function, n_wolves, n_variables, max_iter, lb, ub):
        self.obj_func = objective_function  # Objective function
        self.n_wolves = n_wolves  # Number of wolves
        self.n_variables = n_variables  # Number of variables in the problem
        self.max_iter = max_iter  # Maximum number of iterations
        self.lb = lb  # Lower bound for the search space
        self.ub = ub  # Upper bound for the search space

        self.wolves = np.random.uniform(self.lb, self.ub, (self.n_wolves, self.n_variables))

        self.alpha = np.zeros(self.n_variables)
        self.beta = np.zeros(self.n_variables)
        self.delta = np.zeros(self.n_variables)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")
        self.delta_score = float("inf")

    def update_wolves(self):
        fitness = np.apply_along_axis(self.obj_func, 1, self.wolves)

        sorted_indices = np.argsort(fitness)
        self.wolves = self.wolves[sorted_indices]
        fitness = fitness[sorted_indices]

        # Update alpha, beta, and delta wolves
        self.alpha = self.wolves[0]
        self.beta = self.wolves[1]
        self.delta = self.wolves[2]
        self.alpha_score = fitness[0]
        self.beta_score = fitness[1]
        self.delta_score = fitness[2]

    def optimize(self):
```

```
        for t in range(self.max_iter):
            A = 2 * np.random.random((self.n_wolves, self.n_variables)) - 1  # Random values for
exploration
            C = 2 * np.random.random((self.n_wolves, self.n_variables))  # Random values for
exploitation
            for i in range(self.n_wolves):
                D_alpha = np.abs(C[i] * self.alpha - self.wolves[i])  # Distance to alpha wolf
                D_beta = np.abs(C[i] * self.beta - self.wolves[i])  # Distance to beta wolf
                D_delta = np.abs(C[i] * self.delta - self.wolves[i])  # Distance to delta wolf

                self.wolves[i] = self.alpha - A[i] * D_alpha

                self.wolves[i] = np.clip(self.wolves[i], self.lb, self.ub)

            self.update_wolves()

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

        return self.alpha, self.alpha_score  # Return the best solution found

n_wolves = 30  # Number of wolves
n_variables = 5  # Number of decision variables
max_iter = 100  # Maximum number of iterations
lb = -10  # Lower bound of the search space
ub = 10  # Upper bound of the search space

gwo = GreyWolfOptimizer(objective_function, n_wolves, n_variables, max_iter, lb, ub)
best_solution, best_score = gwo.optimize()
print("Best Solution Found:", best_solution)
print("Best Score:", best_score)
```

Output:

```
Iteration 100/100, Best Score: 1.985808550535119e-30
Best Solution Found: [-4.38373504e-17 -4.54363691e-16 -1.31663573e-15 -2.05502414e-16
  4.09828696e-17]
Best Score: 1.985808550535119e-30
```

Program 6

Parallel Cellular Algorithm

Algorithm:

Lab - 6

Parallel Cellular Algorithm

Algorithm

→ Define a mathematical optimization problem

→ Initiate a grid and population parameters.

→ Implement fitness evaluation, neighbourhood interaction and update rules

→ Iterate to refine solution

→ Output the best solution.

Pseudocode

```
import numpy as np

def fitness_function (grid):
    return -np.sum(grid **2, axis = 2)

def update_states (grid, fitness):
    dirs = [(-1,-1), (-1,0), (-1,1), (0,-1),
            (0,1), (1,-1), (1,0), (1,1)]
    for i in range (grid.shape[0]):
        for j in range (grid.shape[1])
            neighbours = [(i+dx, j+dy) for
            dx, dy in dirs if 0 <=
            i + dx < grid.shape[0]
            grid [i, j] = grid [max (neighbours
            key = lambda n: fitness [n[0], n[1]]
    return grid
```

DATE: PAGE:

```
def parallel_cellular_algorithm (grid
size, space, iteration) :
    grid = np.random.uniform (space [0],
    space [1], (grid_size, grid_size, 2))
    for _ in range (iteration)
        fitness = fitness_function (grid
        grid = update_state (grid, fitnes
    return grid [np.unravel_index
    (np.argmax (fitness), fitness.shape)
    np.max (fitness)

# Example
best_solution, best_fitness = parallel_
cellular_algorithm (10, (-10, 10), 100)
print (f "Best Solution : {best_soluti
Fitness : {best_fitness}")
```

Code:

```
import numpy as np
from multiprocessing import Pool
def update_cell(cell_index, grid, size):
    x, y = cell_index
    neighbors = [
        ((x-1) % size, y), ((x+1) % size, y),
        (x, (y-1) % size), (x, (y+1) % size)
    ]
    new_state = sum(grid[n[0], n[1]] for n in neighbors) % 2  # example: majority rule
    return (x, y, new_state)
def parallel_update(grid, size, num_iterations):
    pool = Pool(processes=4)
    for iteration in range(num_iterations):
        print(f"Iteration {iteration + 1}:")
        indices = [(x, y) for x in range(size) for y in range(size)]
        result = pool.starmap(update_cell, [(i, grid, size) for i in indices])

        for x, y, new_state in result:
            grid[x, y] = new_state
        print(grid)
    return grid
grid_size = 10
grid = np.random.randint(2, size=(grid_size, grid_size))
print("Initial state:")
print(grid)
num_iterations = 2
updated_grid = parallel_update(grid, grid_size, num_iterations)
```

Output:

```
Iteration 1:
[[1 0 0 1]
 [1 0 1 0]
 [1 0 0 1]
 [0 1 0 1]]
Iteration 2:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Program 7

Gene Expression Algorithm

Algorithm:

Lab - 7

Gene Expression Algorithm (GEA)

Algorithm :

→ Define the problem : create an optimization function to maximise or minimize

→ Initialize parameter : Set the population size, number of genes mutation rate, crossover rate

→ Initialize population : Generate random genetic sequence

→ Evaluate fitness

→ Selection : select top sequences based on fitness

→ Crossover : Combine genetic material of selected sequences to produce offspring

→ Mutation : Apply random mutation to offspring to maintain diversity

→ Gene Expression : Convert genetic sequence into function sol^n

→ Repeat : Continue steps 4 - 8 for all generation

→ Output solution : Return the best solution found.

DATE: PAGE:

Pseudocode

```
import numpy as np

def fitness_function(solution):
    return -(solution**2).sum()

def initialize_population(pop_size,
num_genes, space):
    return np.random.uniform
    (space[0], space[1], (pop_size, num_genes))

def gene_expression_algorithm
(pop_size, num_genes, space,
generation, mutation_rate, crossover_rate):
    population = initialise_population
    (pop_size, num_genes, space)
    for _ in range (generation)
    fitness = (-)(population**2).sum(axis=1)
    parents = population[np.argsort
    (fitness)[-int(pop_size*crossover_
    rate):]]
    offspring = parents[np.random.randint
    (len(parents), size = pop_size -
     len(parents))]
     population = np.vstack((parents, offspring))
best_idx = np.arg(-(population**2).
sum(axis=1))
return population[best_idx],
(population[best_idx]**2).sum()
```

Code:

```
import random
import numpy as np
import operator

# Function set and terminal set
FUNCTIONS = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
TERMINALS = ['x', 1, 2, 3, 4]  # x and constants

def random_gene(length=10):
    """Generate a random chromosome (gene)."""
    return [random.choice(list(FUNCTIONS.keys()) + TERMINALS) for _ in range(length)]


def decode_chromosome(chromosome, x):
    """Decode chromosome into a functional expression tree (phenotype)."""
    stack = []
    for gene in chromosome:
        if gene in FUNCTIONS:  # If it's a function, pop arguments and apply
            if len(stack) < 2:  # Avoid errors if stack has fewer than 2 elements
                stack.append(0)
                continue
            b = stack.pop()
            a = stack.pop()
            try:
                result = FUNCTIONS[gene](a, b)
            except ZeroDivisionError:
                result = 1  # Avoid division by zero
            stack.append(result)
        elif gene == 'x':
            stack.append(x)
        else:
            stack.append(gene)
    return stack[0] if stack else 0  # Return top of stack as output


def fitness_function(chromosome, target_function, x_values):
    """Calculate fitness based on Mean Squared Error."""
    predictions = [decode_chromosome(chromosome, x) for x in x_values]
```

```
    targets = [target_function(x) for x in x_values]
    mse = np.mean([(p - t) ** 2 for p, t in zip(predictions, targets)])
    return mse


def selection(population, fitnesses):
    """Select individuals based on fitness (roulette wheel selection)."""
    total_fitness = sum(1 / (f + 1e-6) for f in fitnesses)  # Avoid division by zero
    probabilities = [(1 / (f + 1e-6)) / total_fitness for f in fitnesses]
    return population[np.random.choice(len(population), p=probabilities)]


def mutate(chromosome, mutation_rate=0.1):
    """Apply mutation to a chromosome."""
    new_chromosome = chromosome[:]
    for i in range(len(new_chromosome)):
        if random.random() < mutation_rate:
            new_chromosome[i] = random.choice(list(FUNCTIONS.keys()) + TERMINALS)
    return new_chromosome


def crossover(parent1, parent2):
    """Perform one-point crossover between two parents."""
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2


def gene_expression_algorithm(target_function, x_values, population_size=10, generations=20):
    """Main Gene Expression Algorithm."""
    # Initialize random population
    population = [random_gene() for _ in range(population_size)]

    print("Initial Population:")
    for i, chrom in enumerate(population):
        print(f"Chromosome {i}: {chrom}")

    for generation in range(generations):
        print(f"\nGeneration {generation + 1}:")
        # Calculate fitness for each individual
```

```
        fitnesses = [fitness_function(chrom, target_function, x_values) for chrom in population]
        for i, (chrom, fit) in enumerate(zip(population, fitnesses)):
            print(f"Chromosome {i}: {chrom}, Fitness: {fit:.4f}")

        # Select the next generation
        new_population = []
        for _ in range(population_size // 2):
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population

    # Final results
    print("\nFinal Population and Fitness:")
    fitnesses = [fitness_function(chrom, target_function, x_values) for chrom in population]
    for i, (chrom, fit) in enumerate(zip(population, fitnesses)):
        print(f"Chromosome {i}: {chrom}, Fitness: {fit:.4f}")

    best_index = np.argmin(fitnesses)
    print("\nBest Solution:")
    print(f"Chromosome: {population[best_index]}, Fitness: {fitnesses[best_index]:.4f}")

# Target function for regression
def target_function(x):
    return x**2 + 2*x + 1  # Example: f(x) = x^2 + 2x + 1

# Input values
x_values = np.linspace(-10, 10, 20)

# Run the algorithm
gene_expression_algorithm(target_function, x_values, population_size=10, generations=10)
```

Output:

```
Best Solution:
Chromosome: [1, 3, '+', 2, 1, 4, '*', '*', '*', 3], Fitness: 1259.2067
<ipython-input-3-6df17022c257>:25: RuntimeWarning: divide by zero encountered in scalar divide
  result = FUNCTIONS[gene](a, b)
```