VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Anish Budavi (1BM23CS401)

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 **Anish Arjun Budavi (1BM23CS401),** 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.

Prof. Surabhi S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

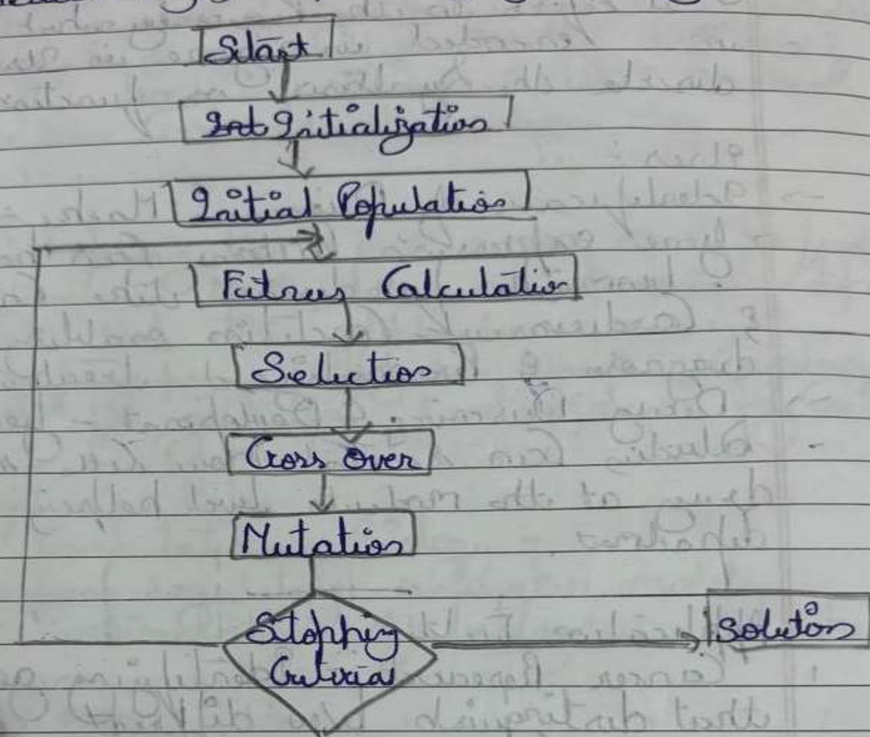
https://github.com/Anishbudavi/ANISH-BUDAVI-1BM23CS401-BIS-LAB.git

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:

Lab 2.
Understanding & Implementing genetic algo



Consider the fn of maximizing
$f(x) = x^2$.

x can vary b/w 0 to 31.
(∵ Permutation upto 5 bits)

min value = 0 max. value = 31
(00000) (11111)

The objective fn has us
$f(x) = x^2$

Step 1:- Selecting Initial Population.
Let's Randomly Select,
Here initial Population of Size 4 is chosen.

Initial Population (randomly chosen) → 01101,
11000,
00100,
10010

Now
Calculate fitness Value

~~SlNo initial Popl X Value Fitness $F(x)=x^2$ Prob %Prob Expected Count~~

SlNo	initial Popl	X Value	fitness $f(x)=x^2$	Prob	%Prob	Expected Count	Actual Count
1	01101	13	169	0.1557	15.57	0.6230	1
2	11000	24	576	0.5308	53.08	2.1235	2
3	00100	4	16	0.0147	1.47	0.0589	0
4	10010	18	324	0.2986	29.86	1.1944	1.
Sum			1085	1.0	100	4.0	4
avg			271.25	0.25	25	1	1
Max			576	0.5308	53.08	2.1235	2.

Expect $\dfrac{F(x)}{Avg(\Sigma Fx)}$

$Prob = \frac{f(a)}{\Sigma f(a)} \Rightarrow \frac{(13)^2}{1085} \rightarrow \frac{(4)^2}{1085}$

$\Rightarrow \frac{(24)^2}{1085} \quad \frac{(18)^2}{1085}$

$Expected\ Count = \frac{f(ai)}{Avg(\Sigma f(a))}$

String no	Mating Pool	Crossover Point	offspring after Crossover	X Value	fitness $f(a) = x^2$
1	01101	32	01100	12	144
2	11000		11001	25	25
3	11000	2	11010	26	676
4	10010		10000	16	256
Sum					1701
avg					425.25
max					676

The fitness Values 676 is highest than earlier 576

Mutation

String no	offspring after Crossover	Mutation chromosome for offspring	offspring after Mutation	x value	fitness $f(a) = x^2$
1	01100	10000	11100	28	784
2	11001	00000	11001	25	625
3	11010	00000	11010	26	676
4	10000	00101	101001	21	441
Sum					2526
avg					631.5
Max					784

784 > 676
Current Max Receiver Max

Complete the algorithm.

Step 2: Filters Calculation & Selection

16.10 Step 3: Cross over
Mating Pool: To form the mating Pool we look into the actual Count of filters. Calculation table & no of Values in actual Column is the number of item That Value will be Used in Mating Pool
Step 4: Mutation of the Crossover.

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

Algorithm:

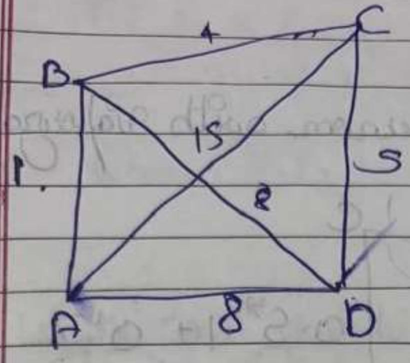
Ant Colony Optimization
Marco Dorigo in 1992
=> The main inspiration of the ACO algorithm
Come from Stigmergy
Stigmergy refers to the interaction & Coordin-
-ation of organism in nature by modifying
the environmental

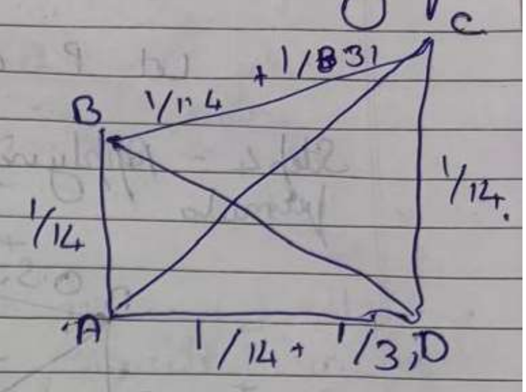
Ant & Stigmergy
-> It Produces chemicals Called Pheromone
-> They use Pheromones to Communicate
this is similar to water in our analogy
They use a very simple technique our analogy
to find the shortest path from the
to a source of food. They make decision
based on Probability

Mathematical Model of ACO

-> Pheromone (Model & Vaporization)
-> Decision Making
Pheromone Matrix
$\Delta \tau^{k}_{ij} = \{ 1/L_K$ K^{th} ant travels on
-> To Calculate the amount of a Pheromone
edge of i, j
$\tau^{k}_{ij} = \sum_{k=1} \Delta \tau^{k}_{ij}$ without Vaporiz-
-ation
$\tau^{k}_{i,j} = (1-\rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta \tau^{k}_{i,j}$ with Vaporization
ρ rau is a Constant

Cost graph. Pheromone graph

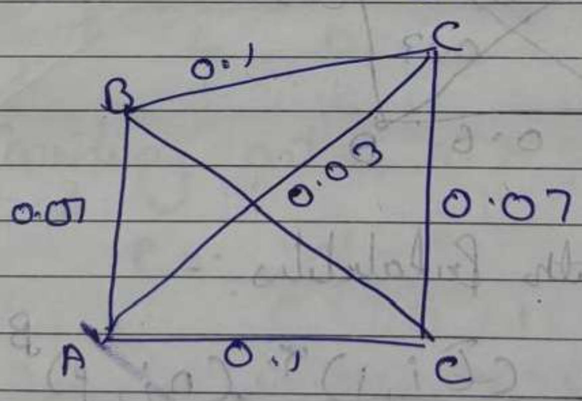




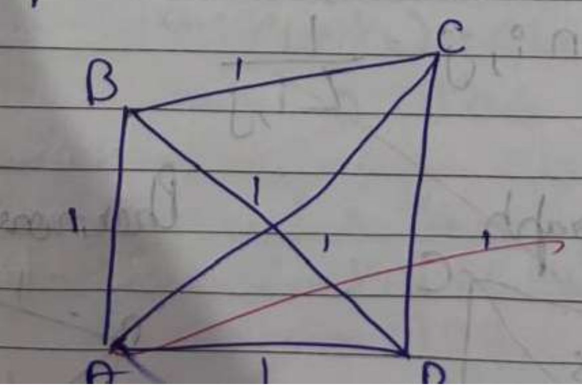
Ans 1 $4 = 14 \rightarrow A[i,j] = 1/14, \tau_{i,j} \sum_{k=1}^{m} \Delta\tau^{k}_{i}$

Ant 2 $= L_2 = 31 \rightarrow \Delta\tau_{i,j} = 1/31$

Step 2: Pheromone graph.



Step 3

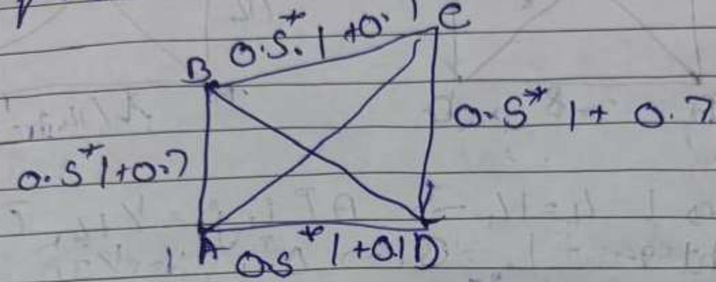


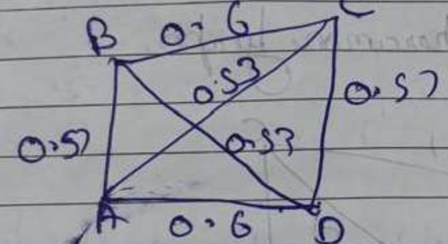
We assume weight is 1 i.e $\tau_{i,j}$

$$\tau_{i,j} = (1-P)\,\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau^{k}_{i,j}$$

Let $P = 0.5$

Step 4: Applying Pheromone with vaporization formula



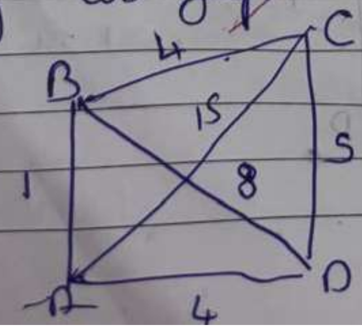


Calculating the Probabilities:-

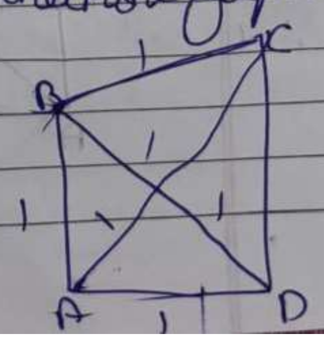
$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}\,(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}$$

When $\eta_{i,j} = \frac{1}{L_{i,j}}$

Eg Cost graph



Pheromone graph

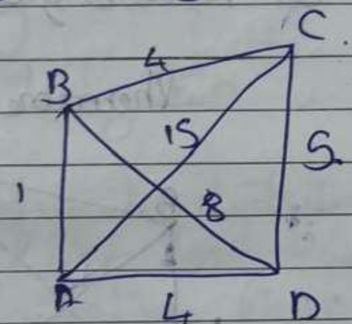


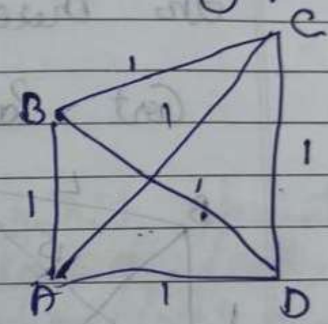
Considering Calculating the Probabilities :-

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha} (\eta_{i,j})^{\beta}}{\Sigma((\tau_{i,j})^{\alpha} (\eta_{i,j})^{\beta})}$$

where $\eta_{i,j} = \frac{1}{L_{i,j}}$ η :-

Eg :- cost graph Pheromone graph



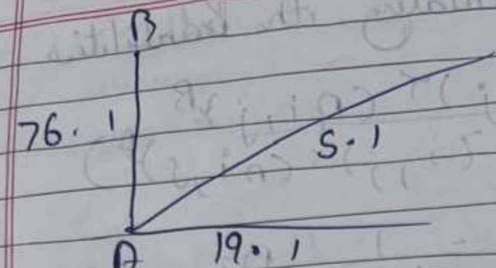


Considering ant is on A & has to make dec

for Edge A, B
$$P = \frac{1 \times 1/1}{(1 \times 1/1) + (1 \times 1/15) + (1 \times 1/4)}$$
$\Rightarrow 0.7595$

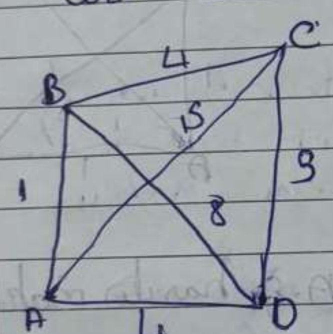
for Edge A, C
$$P = \frac{1 \times 1/15}{(1 \times 1/1) + (1 \times 1/15) + (1 \times 1/4)}$$
$\Rightarrow 0.0506$

for Edge A, D
$$P = \frac{1 \times 1/4}{(1 \times 1/1) + (1 \times 1/18) + (1 \times 1/4)}$$
$\Rightarrow 0.1899$

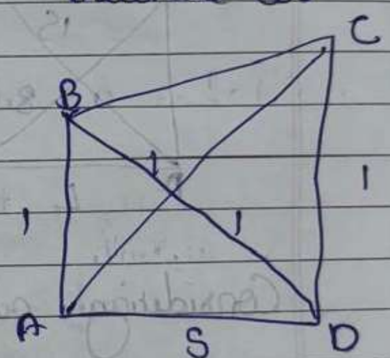


This Show the impact of Cost when we have Same Pheromone in the Leg - We will increase the Pheromone level on A to D & See the impact

Cost Path



Pheromone Cost



A to B $\dfrac{1 \times 1/1}{(1 \times 1/1)\ (1 \times 1/15) + (5 \times 1/4)} \Rightarrow 0.4317 = 43\%$

A to C $\dfrac{1 \times 1/15}{(1 \times 1/1) + (1 \times 1/15) + (5 \times 1/4)} \Rightarrow 0.5396 = 54\%$

Now the Path which will be chosen in A to D Path Solution - It is based on method Called Roulette wheel we use the Selection Path of genetic algorithm

Probabilities | 0.76 | 0.19 | 0.05 |

Cummulative Sum [1 | 0.24 | 0.05]
Now We Select from Cummulative Sum for that We generate a random (r) in [0, 1]

$$\begin{cases} 0.24 < r \le 1.00 & \text{We choose A to B} \\ 0.05 < r \le 0.24 & \text{We choose A to D} \\ 0.06 \le r \le 0.5 & \text{We choose A to C} \end{cases}$$ (graph from Fig 1.)

Algorithm

Initialization.

-> Define the distance matrix which represents the cost b/w each pair of nodes
-> Initialize Pheromone level on all edges uniformly
-> Set Parameters such as the no of ants the no of best to consider, the no of iteration pheromone decay rate & the influence of Pheromone & distance on Path Selection

Iteration :-

Generate Path :

-> For each ant generate a complete Path starting from a designated starting node
-> Keep track of visited nodes to avoid cycles
-> For each node, choose the next node to move to ~~based~~ Pheromone level & distance using a Probability Probabilities
=> Once an ant Completed its Path return

the starting node

=> Evaluate Path:

Calculate the total distance of each path generated by ants.

update Pheromone levels

=> Sort the path based on this distance & Select the best n-best path.

=> For each move in these best paths increase the Pheromone level on that path based on the inverse of the distance

=> Apply Pheromone decay to all paths to simulate evaporation

=> update Shortest paths

Keep track of the shortest path found so far during the iteration

O/P

After completing the iterations return the best path

23.10

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:

Bafna Gold
Date: Page:

PSO:- Particle Swarm optimization
PSO is a Powerfull meta-heuristic optimization algorithm & by Swarm behaviour observed in nature such as birds Schooling It is a Simplified Social System

-> Cognitive Co-efficient :- It influence how much a Particle relies on its best known Particle.

-> Social - Co-efficient :- It determines the influence of the global best Position found by Swarm.

PSO algorithm.

-> Intialise the Particle Population array x =>

=> loop

=> For each Particle Calculate the fitness Using the fitness function $f(x-i)$

=> Compare the Current fitness Value with its best Replace the best with the Current Value $x-i$ if it is better than the best

=> check the Swarm best Particle from individual Particles best & assign the best array to the global best P_g

=> Calculate the Velocity $\{V-i(t+1)\}$ update the Position of the Particle to $x-i(t+1)$

=> If a Criteria is meet exit the loop

=> End loop

Ex. Let Consider we have to find the Minimum Value $f(x, y) = x^2 + y^2$ it has a minimum at Point $(0, 0)$

Initialization: Imagine we have a Swarm Particle each Particle represent a Potential Solz Positioned in a Search Space no of Particle 5.

initial random Particle

P_1, P_2, P_3, P_4, P_5

evaluate fitness:

$P_1 : f(x, y) = x^2 + y^2$

$P_2 = P_3 = P_4 = P_5$

=> update the Personal & global best
Each Particle remember its best Position
Swarm identifies the best Position found by the Particle

=> update Velocities & its Position
based on their own Experience & global best

=> Iterate: Repeat for a Set of iteration
=> Particle explore Space based on their updated Velocities
=> Continuously evaluate the fitness

Convergence: after Several iteration the Particle will cluster around global best Position $(0, 0)$ as the best min far

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:

Cuckoo Search Algorithm (CSA)

CSA is a Metaheuristic optimization algorithm that helps developers finds the best Parameter & Variants for Specific Problem

Levy Flight : Random Walk with steps that follows a levy distribution: It helps the algorithm escape local minima & explore the Search Space more effectively

Advantages

→ Global Search: The Use of Levy Flight allows the algorithm to explore the Search Space broadly avoiding local Minima & improving the chance of finding global optimum

→ Simplicity :- It is Simply to implement with fewer Parameters

→ Application

Image Processing : Image Segmentation edge detection & other optimization tasks in Computer Vision

Data clustering : Finding optimal clusters, clustering problem big data analysis

Algorithm.

→ Initialization: Initialize a Population units (Solution) x_i, where each $x_i \in R^d$ & evaluate their fitness $f(x_i)$.

$f(x) \to$ evaluate the quantity of a given solution x

Main loop: for each iteration from Maxiter repeat the following steps

a] Levy flight: For each nest x_i, generate a new candidate solution x_i using levy flight equation

$$x_i^{new} = x_i + \alpha \, levy(\lambda).$$

Evaluate the fitness $f(x_i^{new})$.

b] Replacement:-
If $f(x_i^{new}) < f(x)$ replace the old ne with new one $x_i = x_i$ new.

c] Nest Abandonment:-
Rank the nests according to their fitness Value
Abandon the worst P_a fraction of nest & replace them with new randomly generated nests

d] Global best Solution:-
Track the solution found so far across all nests of a better solution is found update

Bafna Gold
Date: Page:

the global best solution

$X_{best} = argmin\ f(x_i)$.

Termination:- After MaxIts iterations deliver the global best solution X_{best}

13/11

output

Best solution [-0.07413483 0.16305057
-2.87130585 -1.84049391, 0.09155671]

Best Value (Rastrigin Function): 47.52852537
333939

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:

Grey Wolf optimization

Main Phase:-

- → Tracking, chasing & approaching the Prey
- → Pursuing, encircling, harassing the Prey until its stops moving
- → Attack towards the Prey

It is an Meta-heuristic algorithm that Simulates the leadership & hunting Mechanism of grey wolves in nature

Purpose

- → Solving optimization Problems
- → Mimicking Grey wolf Social Behaviour
- → Global & local Search Balance

Applications

- → Engineering Design optimization
- → AI & ML Feature Selection
- → Image & Signal Processing
  Used for optimizing Parameters in img enhancement / techniques such as edge detection
- → Robotics

Algorithm

1] Intialize Parameters
-> Randomly initialize the Position in the Search Space
-> X_i represent the Position of the i^{th} Wolf
-> Initialize Parameters
$a = 2$ (decrease overtime)
$A = 2a \cdot r_1$ & $c = 2 \cdot r_2$
where r_1 & r_2 are random Vector in

Evaluate Fitness

=> Evaluate the Fitness of each wolf using the objective function $f(x_i)$
=> Identifying the best three Wolves

X_α -> best Fitness
X_β -> Second best
X_δ -> third best

3 Main Loop (For each iteration t from 1 to T
update the different a based on the iteration

$$a = 2 \times \left(1 - \frac{t}{T}\right)$$

It Controls the exploration & exploitation balance
t - increase a - decrease

update Position of Wolves For each wolf i
Calculate the new Position Using Position of 3 best α, β, & δ

=> Calculate distance b/w the each Wolf & each Wolf of leaders

$$D_\alpha = |C_1 \cdot X_\alpha(t) - X_i(t)|,$$
$$D_\beta = |C_2 \cdot X_\beta(t) - X_i(t)|,$$
$$D_\gamma = |C_3 \cdot X_\gamma(t) - X_i(t)|,$$

update the Position of each Wolf

$$X_1(t+1) = X_\alpha(t) - A_1 \cdot D_\alpha$$
$$X_2(t+1) = X_\beta(t) - A_2 \cdot D_\alpha,$$
$$X_3(t+1) = X_\gamma(t) - A_3 \cdot D_\gamma$$

The new Position wolf is the average of the updated Position

$$X_i(t+1) = \frac{X_1(t+1) + X_2(t+1) + X_3(t+1)}{3}$$

=> update the fixtures of all Wolves

=> update the leader (X_α, X_β, X_γ) based on the new fitness Value

7. Termination Condition

=> If the stopping Criteria are Stop the algorithm

8. Return Best Solution

The final Solution is the Position of α wolf X_α, as it represent the best Solution found by the Algorithm.

Output

optimal Solution: [1.33185437e-11, 1.78705732e-11 9.88388617e-12 8.81344255e-12]

optimal Value: 1.7709399328127607e-17

20.11

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:

Parallel Cellular Algorithms

Purpose

Parallel Cellular Algorithms (PCA's) are computational techniques that leverage the concept of cellular automata in parallel computing environments. Cellular automata are mathematical models that consist of a grid of cells each of which evolves based on a set of rules.

Application

1 Scientific Modelling and Simulations
* Weather patterns, population dynamics and traffic flows

2 Image Processing
* Applications include edge detection, noise reduction and segmentation

3 Optimization Problem
* Solving combinatorial problems such as Travelling Salesman Problem

4 Cryptography
* Generating pseudorandom numbers & cryptographic primitives

Date: __/__/__

Algorithm

1. Initialization
* Define a 2D grid $G(t)$ of size $M \times N$ where:

$G_{ij}(t) \in \{0, 1\}, 1 \leq i \leq M, 1 \leq j \leq N$
represents the state of the cell (i, j) at time t.

2. Neighborhood
Define the neighborhood $N_{ij}(t)$ of cell $G_{ij}(t)$ as

$$N_{ij}(t) = \{G_{i+a, j+b}(t) \mid a, b \in \{-1, 0, 1\}, (a, b) \neq (0, 0)\}.$$

3. State update Rule:
* Compute the sum of live neighbor $S_{ij}(t)$:

$$S_{ij}(t) = \sum_{(a,b) \in \{-1,0,1\}, (a,b) \neq (0,0)} G_{i+a, j+b}(t).$$

update the state of the cell. $G_{ij}(t+1)$

4. Parallelization:
* Divide the grid into sub-grid for parallel processing
* Each process computes the updated state $G_{ij}(t+1)$

Date: ___/___/___

5. Synchronization
Exchange boundary data between sub-g[rids] after each iteration to maintain continuity of neighbour states

6. Termination
Repeat steps 3 – 5 for a fixed number of iterations T, or stop when C G (t+1) = [...] indicating a steady state

Output

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

2.7.21

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:

7:- Gene Expression Algorithm

They are inspired by the biological Process of gene expression in living. This Process involves of the translation of genetic information encoded in DNA into functional Products

Algorithm

1] Define the Problem: Create a mathematical function to optimize

2) Initialize Parameter: Set the Population Size, no of genes, mutation rate, Crossover rate & no of generation

3) Initialize Population: generate a initial Population of random genetic Sequence

4] Evaluate Fitness:- Evaluate the fitness of each genetic Sequence based on the optimization Function

5] Selection:- Select genetic Sequence based on their fitness for reproduction

6] Crossover:- Perform Crossover b/w Selected Sequence to Produce offspring

7) Mutation:- Apply Mutation to the offspring to introduce Variability

8 Give Expression :- Translate genetic Sequence into functional

9] Iterate :- Repeat the Selection Crossover, mutation & give expression Process for a fixed-number of generation and until Convergence Criteria are meet

10 Output the best Solution :- Track & output the best Solution found during the iteration

Initial Population :-

Generation :-

chromosome : 0 : 2509.733 /
1 : 2657.1015
2 : 9nf
3 : 2569.7331
4 : 2546.5752

Final :-
Fitness :-
chromosome : 0 : 2582.4173
1 : 2587.4173
2 : 3147.2067
3 : 1259.2067
4 : 1416.6804

Best Fitness :-
chromosome fitness = 1259.2067

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)
```