# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB RECORD**

# Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**Raghavendra R (1BM22CS214)**

***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 **Raghavendra R (1BM22CS214),** 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.

| | |
|---|---|
| SOUMYA T<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Joythi S Nayak<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index



Github Link:
https://github.com/RaghavendraR-CS214/BIS-LAB/tree/main

**Program 1**

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

Algorithm:

LAB-01:

Genetic Algorithm for optimization problems:

code:

```
import random
import numpy as np
def fitness_fuction(x):
    return x**2

population_size = 10
generation = 50
mutation_rate = 0.1
crossover_rate = 0.8

def create_population(size):
    return np.random.uniform(-10,10, size)

def evaluate_fitness(population):
    return np.array([fitness_fuction(ind) for ind in population])

def select_parents(population, fitness):
    total_fitness = np.sum(fitness)
    selection_probs = fitness / total_fitness
    return population[np.random.choice(len(population), size=2, p=selection_probs)]

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        alpha = random.random()
        child = alpha*parent1 + (1- alpha)*parent2
        return child
    return parent1

def mutate(child):
    if random.random() < mutation_rate:
        mutation_point = random.uniform(-10, 10)
        return mutation_point
    return child.
```

```
def genetic_algorithm():
    population = create_population(population_size)
    best_overall_fitness = np.inf
    best_overall_individual = None
    best_gen = -1
    for gen in range(generations):
        fitness = evaluate_fitness(population)
        best_fit = np.max(fitness)
        best_indi = population[np.argmax(fitness)]
        if best_fit > best_overall_fitness:
            best_overall_fitness = best_fit
            best_overall_indi = best_indi
            best_generation = gen
        new_pop = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population, fitness)
            child1 = crossover(parent1, parent2)
            child2 = crossover(parent2, parent1)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_pop.extend([child1, child2])
        population = np.array(new_pop)
    return best_overall, best_fitness, best_overall_fit,
        best_gen

best_solution, best_fitness, best_gen = genetic_algo()
print(f"Best Solution: {best_solution} with fitness:{best_fitness} at
    generation : {best_gen}")
```

o/p:
Best Solution: -9.89748 5096 with Fitness 97.96021123
at generation : 26

Code:

```
import random

# Define the fitness function
def fitness_function(x):
    return x ** 2
```

```python
# Generate initial population
def generate_population(size, lower_bound, upper_bound):
    return [random.uniform(lower_bound, upper_bound) for _ in range(size)]

# Selection - select individuals based on fitness
def selection(population, fitness_values):
    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    selected = random.choices(population, weights=probabilities, k=len(population))
    return selected

# Crossover - create new offspring by combining parents
def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        alpha = random.random()
        child1 = alpha * parent1 + (1 - alpha) * parent2
        child2 = alpha * parent2 + (1 - alpha) * parent1
        return child1, child2
    else:
        return parent1, parent2

# Mutation - introduce random variations
def mutate(individual, mutation_rate, lower_bound, upper_bound):
    if random.random() < mutation_rate:
        individual += random.uniform(-1, 1)
        individual = max(lower_bound, min(upper_bound, individual))  # Keep within bounds
    return individual

# Genetic Algorithm
def genetic_algorithm(population_size, lower_bound, upper_bound, generations, mutation_rate, crossover_rate):
    population = generate_population(population_size, lower_bound, upper_bound)

    for generation in range(generations):
        # Evaluate fitness
        fitness_values = [fitness_function(ind) for ind in population]

        # Selection
```

```
        selected_population = selection(population, fitness_values)

        # Crossover
        next_generation = []
        for i in range(0, len(selected_population), 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1 if i + 1 < len(selected_population) else 0]
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            next_generation.extend([child1, child2])

        # Mutation
        population = [mutate(ind, mutation_rate, lower_bound, upper_bound) for ind in next_generation]

        # Log best fitness of the generation
        best_fitness = max(fitness_values)
        # print(f"Generation {generation + 1}: Best Fitness = {best_fitness:.4f}")

    # Return the best fitness value from the final generation
    return max(fitness_function(ind) for ind in population)

# Parameters
population_size = 10
lower_bound = -10
upper_bound = 10
generations = 50
mutation_rate = 0.1
crossover_rate = 0.8
print("Raghavendra R, 1BM22CS214")
# Run Genetic Algorithm
best_fitness = genetic_algorithm(population_size, lower_bound, upper_bound, generations, mutation_rate, crossover_rate)
print(f"Best fitness found: {best_fitness:.4f}")
```

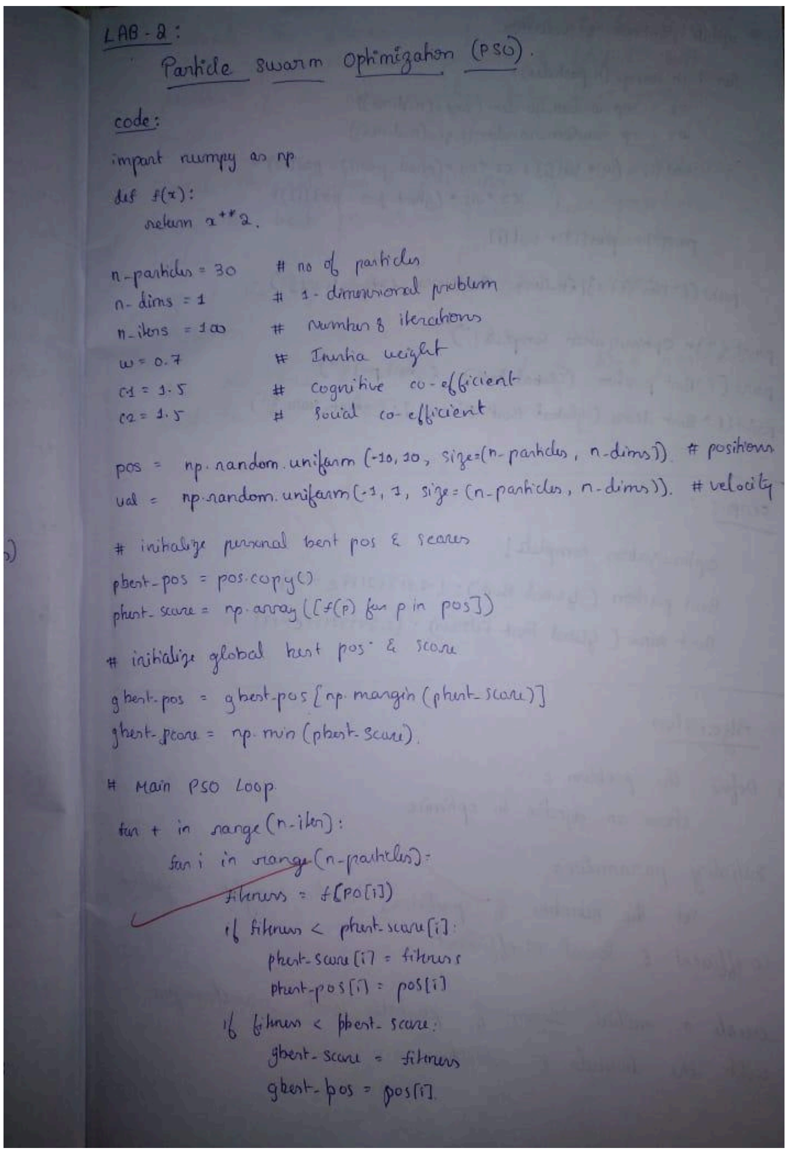
```
PS C:\Users\Dell\Desktop\BIS> & C:/Users/Dell/AppData
enetic.py
Raghavendra R, 1BM22CS214
Best fitness found: 100.0000
```

**Program 2**

**Particle Swarm Optimization for Function Optimization:**

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

Algorithm:



LAB - 2 :

Particle swarm Optimization (PSO).

code:

```
import numpy as np
def f(x):
    return x**2.

n-particles = 30      # no of particles
n- dims = 1           # 1 - dimensional problem
n_iters = 100         # number of iterations
w = 0.7               # Inertia weight
c1 = 1.5              # cognitive co-efficient
c2 = 1.5              # social co-efficient

pos = np.random.uniform (-10, 10, size=(n-particles, n-dims)) # positions
val = np.random.uniform(-1, 1, size=(n-particles, n-dims)). # velocity

# initialize personal best pos & scores
pbest-pos = pos.copy()
pbest- score = np.array([f(p) for p in pos])

# initialize global best pos & score
gbest-pos = gbest-pos[np.margin(pbest-score)]
gbest-score = np.min(pbest-score).

# Main PSO Loop
for t in range(n-iter):
    for i in range(n-particles):
        fitness = f[PO[i])
        if fitness < pbest-score[i]:
            pbest-score[i] = fitness
            pbest-pos[i] = pos[i]
        if fitness < pbest- score:
            gbest- score = fitness
            gbest- pos = pos[i]
```

# update positions & velocities.

```
for i in range(n_particles):
    r1 = np.random.random(size=(n_dims))
    r2 = np.random.random(size=(n_dims))
    val[i] = (w * vel[i] + c1 * r1 * (pbest_pos[i] - pos[i]) +
                         c2 * r2 * (gbest_pos - pos[i]))

    pos[i] = pos[i] + val[i]

  print(f"Iter {t+1}/{n_iter}, Bestscore: {gbest_score}")

print("\n Optimization Complete!")
print(f"Best position (Global best): {gbest_pos}")
print(f"Best Score (Global Best fitness): {gbest_score}")
```

OUTPUT :

Optimization complete!

Best position ( Global Best): [-2.81312189e-05]

Best score ( Global Best Fitness) : [1.84542157e-13]

→ Algorithm:

1) Define the problem :
   choose an objective to optimise

2) initialize parameter :
   set the number of particles, inertia weight, cognitive co-efficient & Social co-efficient

3) create a initial swarm of particles with random pos with the bounds & velocities.

4) Evaluate the fitness using the objective function ($f(x)$), update the best position & global best position, if the new position better than previous position

5) Update the velocities & positions: update a velocity based on best position & global best position. update a position using position & velocity repeat the step until it reaches the max iterations

6) After the final iterations, print a best positions and the best value

Code:

```python
import random

def objective_function(x):
    return sum(x_i ** 2 for x_i in x)

class Particle:
    def __init__(self, dimension, bounds):
        self.position = [random.uniform(bounds[0], bounds[1]) for _ in
range(dimension)]
        self.velocity = [random.uniform(-1, 1) for _ in range(dimension)]
        self.pBest = list(self.position)
        self.pBest_fitness = objective_function(self.position)

class PSO:
    def __init__(self, dimension, bounds, num_particles=30,
max_iterations=100):
        self.dimension = dimension
        self.bounds = bounds
        self.num_particles = num_particles
```

```
        self.max_iterations = max_iterations
        self.particles = [Particle(dimension, bounds) for _ in
range(num_particles)]
        self.gBest = list(self.particles[0].position)
        self.gBest_fitness = self.particles[0].pBest_fitness
        self.w = 0.5
        self.c1 = 1.5
        self.c2 = 1.5

    def optimize(self):
        for iteration in range(self.max_iterations):
            for particle in self.particles:
                fitness = objective_function(particle.position)

                # Update personal best (pBest)
                if fitness < particle.pBest_fitness:
                    particle.pBest = list(particle.position)
                    particle.pBest_fitness = fitness

                # Update global best (gBest)
                if fitness < self.gBest_fitness:
                    self.gBest = list(particle.position)
                    self.gBest_fitness = fitness

            # Update velocity and position for each particle
            for particle in self.particles:
                for i in range(self.dimension):
                    # Update velocity
                    r1, r2 = random.random(), random.random()
                    particle.velocity[i] = (self.w * particle.velocity[i]
                                            + self.c1 * r1 *
(particle.pBest[i] - particle.position[i])
                                            + self.c2 * r2 * (self.gBest[i] -
particle.position[i]))
                    # Update position
                    particle.position[i] += particle.velocity[i]

                    # Ensure position stays within bounds
                    particle.position[i] = max(self.bounds[0],
min(particle.position[i], self.bounds[1]))

        return self.gBest, self.gBest_fitness
```

```python
# Define parameters
dimension = 2  # Number of dimensions
bounds = (-10, 10)  # Search space bounds for each dimension
num_particles = 30  # Number of particles in the swarm
max_iterations = 100  # Maximum number of iterations
print('Raghavendra R, 1BM22CS214')

# Create PSO instance and optimize
pso = PSO(dimension, bounds, num_particles, max_iterations)
best_position, best_fitness = pso.optimize()

# Output the result
print(f"Best Position: {best_position}")
print(f"Best Fitness: {best_fitness}")
```

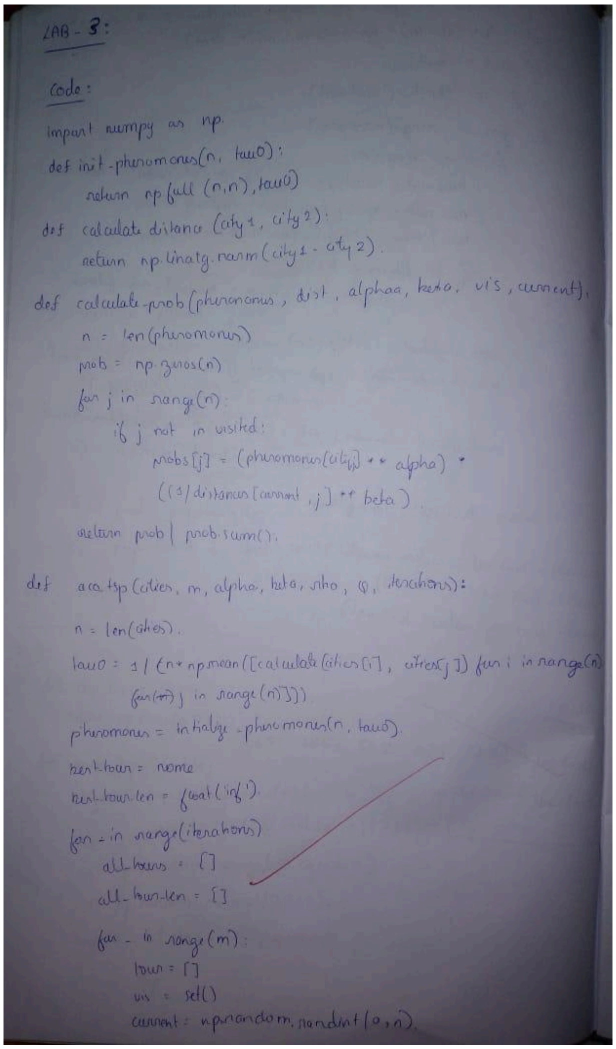
```
PS C:\Users\Dell\Desktop\BIS> & C:/Users/Dell/AppData/Local/Programs/Pytho
so.py
Raghavendra R, 1BM22CS214
Best Position: [-2.3539251665798946e-13, -3.852646973375009e-12]
Best Fitness: 1.4898298338354198e-23
```

**Program 3**

**Ant Colony Optimization for the Traveling Salesman Problem:**

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

Algorithm:



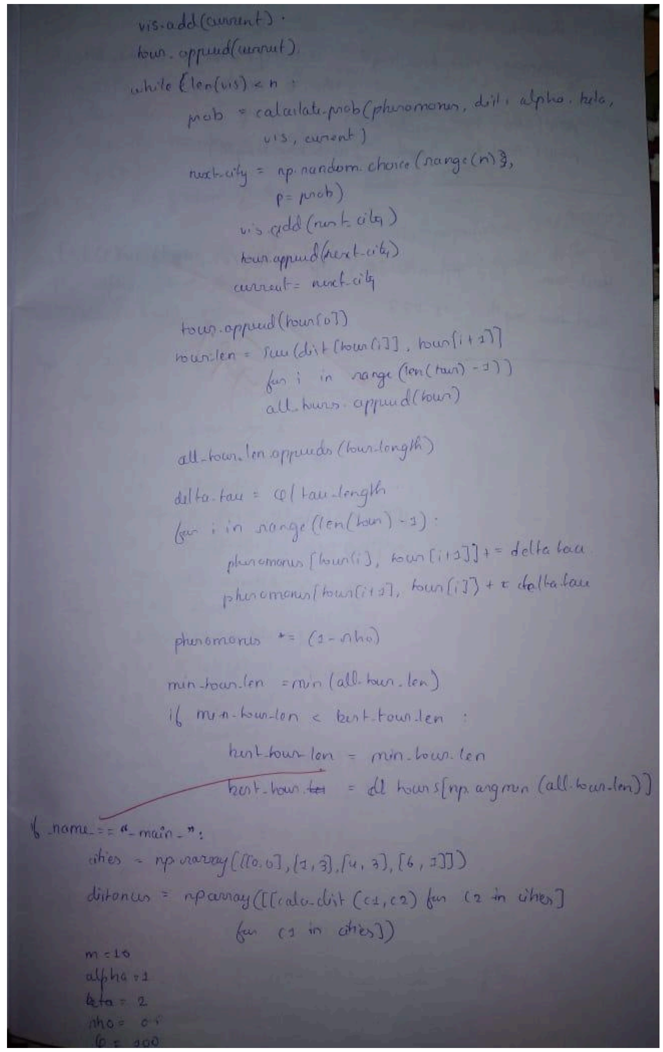
LAB - 3:

Code:

```
import numpy as np.
def init-phenomones(n, tau0):
    return np.full (n,n), tau0)
def calculate distance (city1, city2):
    return np.linalg.norm(city1 - city2).
def calculate-prob(phenomones, dist, alphaa, beta, vis, current).
    n = len(phenomones)
    prob = np.zeros(n)
    for j in range(n):
        if j not in visited:
            probs[j] = (phenomones[city][j] ** alpha) *
            ((1/distances[current, j] ** beta)
    return prob / prob.sum().

def aco tsp (cities, m, alpha, beta, rho, Q, iterations):
    n = len(cities).
    tau0 = 1 / (n * np.mean([calculate (cities[i], cities[j]) for i in range(n)
            for j in range(n)]))
    phenomones = initialize-phenomones(n, tau0).
    best-tour = none
    best-tour len = float('inf').
    for _ in range(iterations)
        all-tours = []
        all-tour-len = []

        for _ in range(m):
            tour = []
            vis = set()
            current = np.random.randint(0, n).
```



```
            vis.add(current).
            tour.append(current).
            while (len(vis) < n :
                prob = calculate-prob(phenomones, dist, alpha, beta,
                        vis, current)
                next-city = np.random.choice(range(n)),
                        p= prob)
                vis.add(next-city)
                tour.append(next-city)
                current= next-city

            tour.append(tour[0])
            tour-len = sum(dist[tour[i]], tour[i+1]]
                    for i in range(len(tour) - 1))
                    all-tours.append(tour)

            all-tour-len.appends(tour-length)

            delta-tau = Q / tau-length
            for i in range(len(tour) - 1):
                phenomones[tour[i], tour[i+1]] += delta tau
                phenomones[tour[i+1], tour[i]] += delta tau

            phenomones *= (1 - rho)

            min-tour-len = min(all-tour-len)
            if min-tour-len < best-tour-len :
                best-tour len = min-tour-len
                best-tour ~~len~~ = all tours[np.argmin(all-tour-len)]

if _name_ == "_main_":
    cities = np.array([[0, 0], [1, 3], [4, 3], [6, 1]])
    distances = np.array([[calc-dist (c1, c2) for c2 in cities]
            for c1 in cities])
    m = 10
    alpha = 1
    beta = 2
    rho = 0.1
    Q = 100
```

Code:

```
import numpy as np
import random

class ACO:
    def __init__(self, n_ants, n_iterations, alpha, beta, rho, deposit,
cities):
        self.n_ants = n_ants  # Number of ants
        self.n_iterations = n_iterations  # Number of iterations
        self.alpha = alpha  # Pheromone influence
        self.beta = beta  # Distance influence
        self.rho = rho  # Evaporation rate
        self.deposit = deposit  # Pheromone deposit constant
```

```
        self.cities = cities  # Coordinates of cities
        self.n_cities = len(cities)  # Number of cities
        self.distances = self.calculate_distances()  # Distance matrix
        self.pheromones = np.ones((self.n_cities, self.n_cities))  # Initial pheromones

    def calculate_distances(self):
        """Calculate Euclidean distances between cities."""
        distances = np.zeros((self.n_cities, self.n_cities))
        for i in range(self.n_cities):
            for j in range(self.n_cities):
                distances[i][j] = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
        return distances

    def construct_solution(self):
        """Construct a solution (path) for one ant."""
        path = [random.randint(0, self.n_cities - 1)]  # Random starting city
        while len(path) < self.n_cities:
            current_city = path[-1]
            next_city = self.choose_next_city(current_city, path)
            path.append(next_city)
        return path

    def choose_next_city(self, current_city, path):
        """Choose the next city based on pheromone and distance."""
        probabilities = []
        for next_city in range(self.n_cities):
            if next_city not in path:
                pheromone = self.pheromones[current_city][next_city] ** self.alpha
                visibility = (1 / self.distances[current_city][next_city]) ** self.beta
                probabilities.append(pheromone * visibility)
            else:
                probabilities.append(0)
        probabilities = np.array(probabilities) / sum(probabilities)
        return np.random.choice(range(self.n_cities), p=probabilities)

    def calculate_distance(self, path):
        """Calculate the total distance of a path."""
        return sum(self.distances[path[i - 1]][path[i]] for i in
```

```python
range(len(path)))

    def update_pheromones(self, paths, distances):
        """Update pheromones based on the paths taken by ants."""
        self.pheromones *= (1 - self.rho)  # Evaporation
        for path, distance in zip(paths, distances):
            pheromone_contribution = self.deposit / distance
            for i in range(len(path)):
                self.pheromones[path[i - 1]][path[i]] +=
pheromone_contribution

    def run(self):
        """Run the ACO algorithm."""
        best_path = None
        best_distance = float('inf')
        for _ in range(self.n_iterations):
            paths = [self.construct_solution() for _ in range(self.n_ants)]
            distances = [self.calculate_distance(path) for path in paths]
            self.update_pheromones(paths, distances)
            # Update the global best path
            for path, distance in zip(paths, distances):
                if distance < best_distance:
                    best_path, best_distance = path, distance
        return best_path, best_distance


# Example Usage
if __name__ == "__main__":
    # Random city coordinates
    cities = [(0, 0), (2, 3), (5, 5), (8, 1)]
    aco = ACO(n_ants=10, n_iterations=100, alpha=1, beta=2, rho=0.1,
deposit=100, cities=cities)
    best_path, best_distance = aco.run()
    print("Raghavendra R, 1BM22CS214")
    print("Best Path:", best_path)
    print("Best Distance:", best_distance)
```

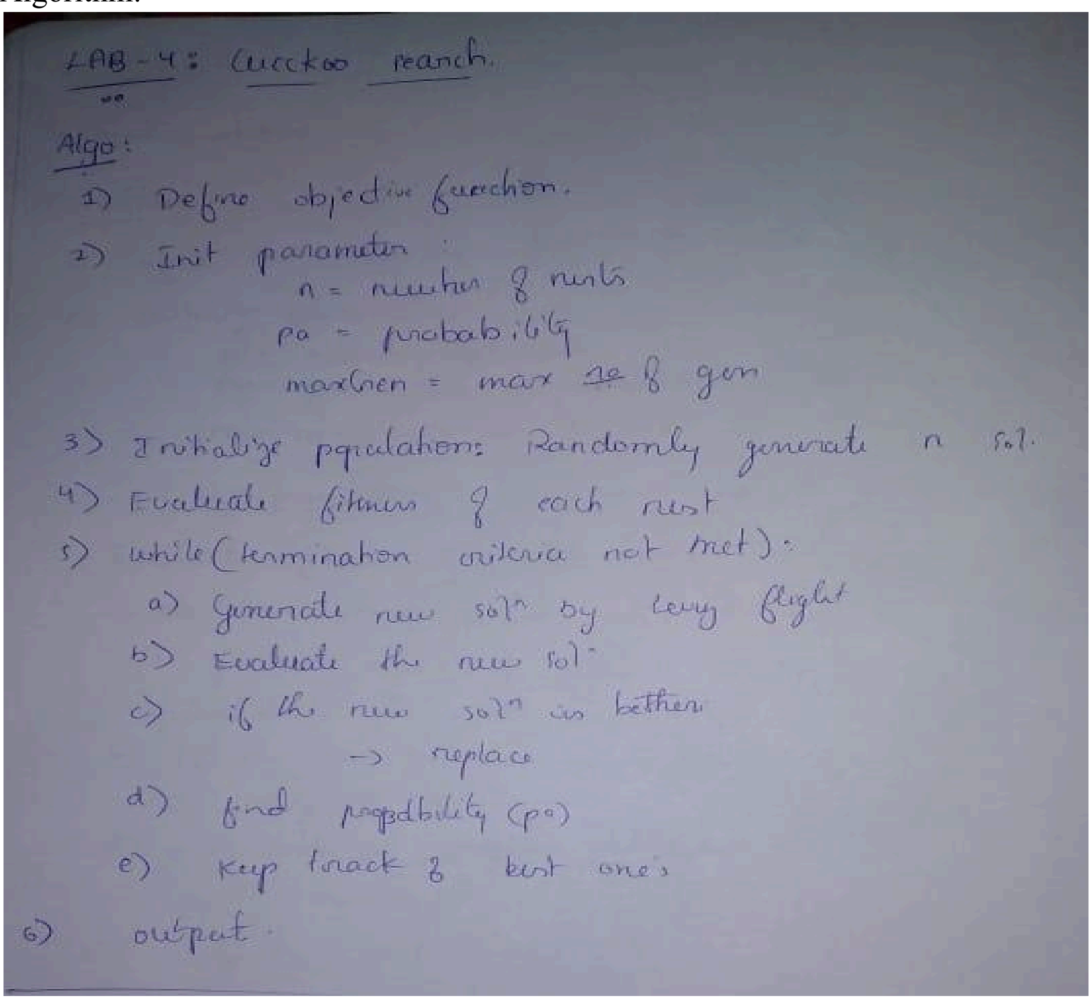
```
Raghavendra R, 1BM22CS214
Best Path: [3, 2, 1, 0]
Best Distance: 20.273360299226525
```

**<u>Program 4</u>**

**Cuckoo Search (CS):**

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

Algorithm:



LAB-4: Cuckoo search.

Algo:

1) Define objective function.
2) Init parameter:
   n = number of nests
   pa = probability
   maxGen = max no of gen
3) Initialize population: Randomly generate n sol.
4) Evaluate fitness of each nest
5) while(termination criteria not met):
   a) Generate new soln by Levy flight
   b) Evaluate the new soln
   c) if the new soln is better
      -> replace
   d) find probability (pa)
   e) Keep track of best one's
6) output.

Code:

```
import numpy as np
import math

# Objective function (example: Sphere function, you can replace it)
def objective_function(x):
    return sum(x**2)  # Minimize the sum of squares


def levy_flight(beta, d):
    sigma_u = (math.gamma(1 + beta) * math.sin(math.pi * beta / 2) /
               (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) /
2)))**(1 / beta)
    u = np.random.normal(0, sigma_u, d)  # Draw from Gaussian distribution
    v = np.random.normal(0, 1, d)
    step = u / (abs(v)**(1 / beta))
    return step

# Cuckoo Search Algorithm
def cuckoo_search(n, d, alpha, pa, maxGen):
    # n: Population size, d: Dimension of the problem
    # alpha: Step size, pa: Discovery probability, maxGen: Max iterations


    nests = np.random.uniform(-10, 10, (n, d))
    fitness = np.array([objective_function(nest) for nest in nests])


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

    beta = 1.5

    # Step 2: Iterative loop
    for gen in range(maxGen):
        for i in range(n):
            # Generate a new solution via Lévy flight
            step = levy_flight(beta, d)
            new_nest = nests[i] + alpha * step * (nests[i] - best_nest)
            new_nest = np.clip(new_nest, -10, 10)  # Keep solutions within
bounds
```

```python
            # Evaluate new fitness
            new_fitness = objective_function(new_nest)
            if new_fitness < fitness[i]:  # Replace with better solution
                nests[i] = new_nest
                fitness[i] = new_fitness

        # Abandon some nests with a probability pa
        for i in range(n):
            if np.random.rand() < pa:
               # Replace with new random solution
                nests[i] = np.random.uniform(-10, 10, d)
                fitness[i] = objective_function(nests[i])

        # Update the current best
        best_nest_index = np.argmin(fitness)
        if fitness[best_nest_index] < best_fitness:
            best_nest = nests[best_nest_index]
            best_fitness = fitness[best_nest_index]

        # print(f"Generation {gen+1}, Best Fitness: {best_fitness:.5f}")
    return best_nest, best_fitness
n = 25
d = 5
alpha = 0.01
pa = 0.25
maxGen = 100
print('Raghavendra R, 1BM22CS214')
best_solution, best_value = cuckoo_search(n, d, alpha, pa, maxGen)
print("Best Solution:", best_solution)
print("Best Fitness Value:", best_value)
```

```python
print('Raghavendra R, 1BM22CS214')
best_solution, best_value = cuckoo_search(n, d, alpha, pa, maxGen)
print("Best Solution:", best_solution)
print("Best Fitness Value:", best_value)
```

```
Raghavendra R, 1BM22CS214
Best Solution: [ 2.99476164 -5.96167039  1.09747839  4.21310143 -8.73101689]
Best Fitness Value: 16.05788538276054
```

**Program 5**

**Grey Wolf Optimizer (GWO):**

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

Algorithm:

LAB-5: Grey wolf

Algorithm:

→ Initialize the population of wolves (positions) randomly within the search space.

→ Define the maximum number of iterations (T) and population size (N)

→ Define the fitness function to evaluate soln.

Evaluate the fitness of each wolf in the population. Identify the alpha (best solution), beta (second-best) and delta (third-best) wolves

```
For t = 1 to T:
    For each wolf i in population:
        For each dimension d:
            A1 = 2*a*rand() - a
            C1 = 2*rand()
            D_alpha = [C1*alpha[d] - x_i[d]]
            X1 = x_alpha[d] - A1*D_alpha
            A2 = 2*a*rand() - a
            C2 = 2*rand()
            D_beta = C2*x_beta[d] - x_i[d]
            X2 = x_beta[d] - A2*D_beta
            A3 = 2*a*rand() - a
            C3 = 2*rand()
            D_delta = C3*x_delta(d) - x_i[d]
            X3 = x_delta(d) - A3*D_delta
            x_i[d] = (x1 + x2 + x3)/3
        End for
    end for
    a = 2 - (2*t/T)
    update alpha, beta & delta wolves based on fitness
```

Code:

```
import numpy as np

def objective_function(x):
    return x ** 2  # The function to minimize

def initialize_wolves(num_wolves, search_space):
    return np.random.uniform(search_space[0], search_space[1], num_wolves)

def update_position(alpha, beta, delta, wolf, a):
    r1, r2 = np.random.rand(), np.random.rand()
    A = 2 * a * r1 - a
    C = 2 * r2
    D = abs(C * alpha - wolf)
    X1 = alpha - A * D

    r1, r2 = np.random.rand(), np.random.rand()
    A = 2 * a * r1 - a
    C = 2 * r2
    D = abs(C * beta - wolf)
    X2 = beta - A * D

    r1, r2 = np.random.rand(), np.random.rand()
    A = 2 * a * r1 - a
    C = 2 * r2
    D = abs(C * delta - wolf)
    X3 = delta - A * D

    return (X1 + X2 + X3) / 3

def grey_wolf_optimization(obj_func, num_wolves=5, max_iter=50,
search_space=(-10, 10)):
    # Initialize wolves' positions
    wolves = initialize_wolves(num_wolves, search_space)
    fitness = np.array([obj_func(wolf) for wolf in wolves])

    # Identify alpha, beta, delta
    sorted_indices = np.argsort(fitness)
    alpha, beta, delta = wolves[sorted_indices[0]],
wolves[sorted_indices[1]], wolves[sorted_indices[2]]
```

```python
    a = 2  # Initial value for the parameter a

    for iteration in range(max_iter):
        for i in range(num_wolves):
            wolves[i] = update_position(alpha, beta, delta, wolves[i], a)
            wolves[i] = np.clip(wolves[i], search_space[0], search_space[1])
# Ensure wolves stay within bounds

        # Recalculate fitness and update alpha, beta, delta
        fitness = np.array([obj_func(wolf) for wolf in wolves])
        sorted_indices = np.argsort(fitness)
        alpha, beta, delta = wolves[sorted_indices[0]],
wolves[sorted_indices[1]], wolves[sorted_indices[2]]

        # Decrease a linearly
        a = 2 - (2 * (iteration / max_iter))

        # print(f"Iteration {iteration+1}: Alpha = {alpha}, Fitness =
{obj_func(alpha)}")

    return alpha, obj_func(alpha)

# Run the algorithm
print("Raghavendra R, 1BM22CS214")
best_position, best_fitness = grey_wolf_optimization(objective_function)
print(f"Best Position: {best_position}")
print(f"Best Fitness: {best_fitness}")
```

```python
# Run the algorithm
print("Raghavendra R, 1BM22CS214")
best_position, best_fitness = grey_wolf_optimization(objective_function)
print(f"Best Position: {best_position}")
print(f"Best Fitness: {best_fitness}")
```

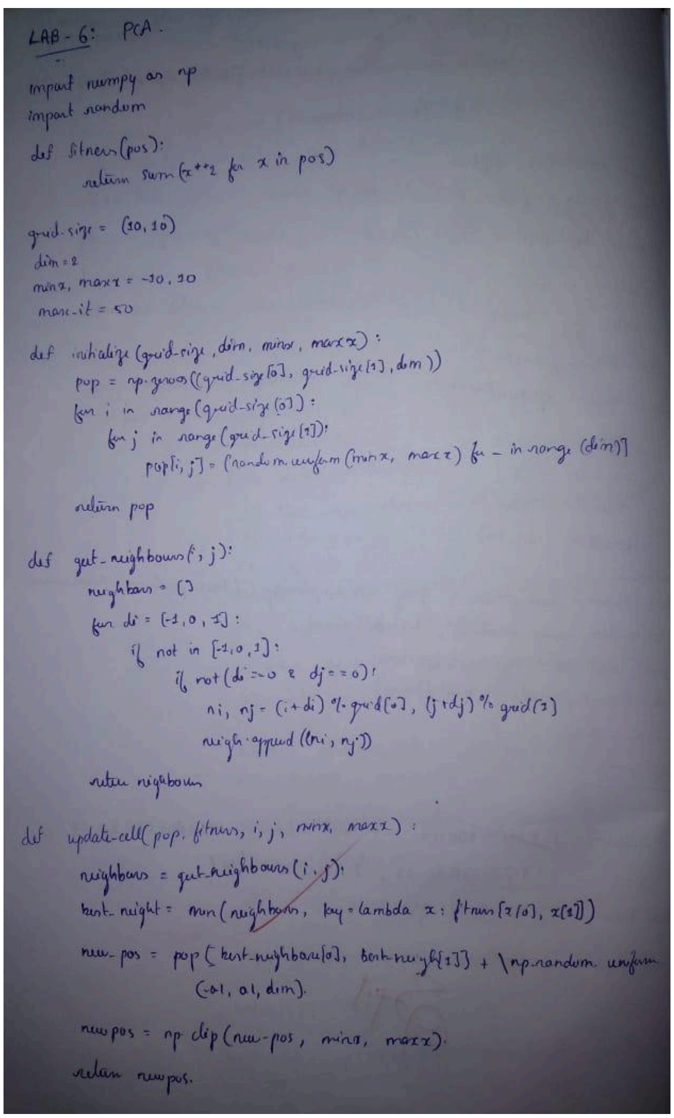
```
Raghavendra R, 1BM22CS214
Best Position: 6.50615910073754e-05
Best Fitness: 4.233010624410991e-09
```

**<u>Program 6</u>**

**Parallel Cellular Algorithms and Programs:**

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

Algorithm:



LAB - 6: PCA.

```
import numpy as np
import random

def fitness(pos):
    return sum(x**2 for x in pos)

grid_size = (10, 10)
dim = 2
minx, maxx = -10, 10
max_it = 50

def initialize(grid_size, dim, minx, maxx):
    pop = np.zeros((grid_size[0], grid_size[1], dim))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            pop[i, j] = [random.uniform(minx, maxx) for _ in range(dim)]

    return pop

def get_neighbours(i, j):
    neighbours = []
    for di = [-1, 0, 1]:
        if not in [-1, 0, 1]:
            if not (di == 0 & dj == 0):
                ni, nj = (i+di) % grid[0], (j+dj) % grid[1]
                neigh.append((ni, nj))

    return neighbours

def update_cell(pop, fitness, i, j, minx, maxx):
    neighbours = get_neighbours(i, j)
    best_neigh = min(neighbours, key = lambda x: fitness[x[0], x[1]])
    new_pos = pop[best_neighbour[0], best_neigh[1]] + \np.random.uniform
                (-0.1, 0.1, dim).

    new_pos = np.clip(new_pos, minx, maxx).
    return new_pos.
```

```
best_index = np.unravel_index(np.argmin(fitness), fitness_grid.shape)
best_pos = population[best_index[0], best_index[1]]
best_fitness = np.min(fitness_grid)
print ("Raghavendra R, 1BM22CS214")
print(best_pos)
print (best_fitness)
```

OUTPUT:

Best solution = [-0.5305, -0.4244]

Best - fitness = 0.200245

Code:

```
import numpy as np
import random

# Step 1: Define the Problem (Optimization Function)
def fitness_function(position):
    """Example fitness function: Sphere function"""
    return sum(x**2 for x in position)

# Step 2: Initialize Parameters
grid_size = (10, 10)  # Grid size (10x10 cells)
dim = 2  # Dimensionality of each cell's position
minx, maxx = -10.0, 10.0  # Search space bounds
max_iterations = 50  # Number of iterations

# Step 3: Initialize Population (Random positions)
def initialize_population(grid_size, dim, minx, maxx):
    population = np.zeros((grid_size[0], grid_size[1], dim))
```

```
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            population[i, j] = [random.uniform(minx, maxx) for _ in
range(dim)]
    return population

# Step 4: Evaluate Fitness (Calculate fitness for each cell)
def evaluate_fitness(population):
    fitness_grid = np.zeros((grid_size[0], grid_size[1]))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            fitness_grid[i, j] = fitness_function(population[i, j])
    return fitness_grid

# Step 5: Update States (Update each cell based on its neighbors)
def get_neighbors(i, j):
    """Returns the coordinates of neighboring cells."""
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if not (di == 0 and dj == 0):  # Exclude the cell itself
                ni, nj = (i + di) % grid_size[0], (j + dj) % grid_size[1]
                neighbors.append((ni, nj))
    return neighbors

def update_cell(population, fitness_grid, i, j, minx, maxx):
    """Update the state of a cell based on the average state of its
neighbors."""
    neighbors = get_neighbors(i, j)
    best_neighbor = min(neighbors, key=lambda x: fitness_grid[x[0], x[1]])

    # Update cell position to move towards the best neighbor's position
    new_position = population[best_neighbor[0], best_neighbor[1]] + \
                   np.random.uniform(-0.1, 0.1, dim)   # Small random
perturbation

    # Ensure the new position stays within bounds
    new_position = np.clip(new_position, minx, maxx)
    return new_position

# Step 6: Iterate (Repeat for a fixed number of iterations)
population = initialize_population(grid_size, dim, minx, maxx)
```

```python
for iteration in range(max_iterations):
    fitness_grid = evaluate_fitness(population)

    # Update each cell in parallel (simultaneously)
    new_population = np.zeros_like(population)
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            new_population[i, j] = update_cell(population, fitness_grid, i,
j, minx, maxx)

    population = new_population

    # Print best fitness at each iteration
    best_fitness = np.min(fitness_grid)
    # print(f"Iteration {iteration + 1}, Best Fitness: {best_fitness}")

# Step 7: Output the Best Solution
best_index = np.unravel_index(np.argmin(fitness_grid), fitness_grid.shape)
best_position = population[best_index[0], best_index[1]]
best_fitness = np.min(fitness_grid)
print("Raghavendra R,1BM22CS214")
print("Best Position Found:", best_position)
print("Best Fitness Found:", best_fitness)
```

```python
print("Raghavendra R, 1BM22CS214")
print("Best Position Found:", best_position)
print("Best Fitness Found:", best_fitness)
```

```
Raghavendra R, 1BM22CS214
Best Position Found: [-0.01963782  0.04692507]
Best Fitness Found: 0.00026949831673197134
```

## Program 7

**Optimization via Gene Expression Algorithms:**

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

Algorithm:

LAB-7:

```
import random
import operator
import math

POPULATION-size = 100
 gen = 5
 rate = 0.1
 crate = 0.7
 max_tree = 5
Function = ['+', '*', '/']
terminal = ['x', '1', '2', '3']

class individual:
    def __init__(self, expression):
        self.expression = expression
        self.fitness = float('inf')

    def evaluate(self, x-val):
        try:
            expr = self.expression.replace('x', str(x-val))
            self.fitness = eval(expr)
        except Exception as e:
            self.fitness = float('inf')

def generate():
    exp = generate-random(max_tree)
    return individual(expr):

def generate-random-(depth):
    if depth == 0 or random.random() < 0.3:
        return random.choice(terminals)
    else:
        return f"{left} {function} {right}".
```

```
def run-gp-algo():
    pop = [generate random] for _ in range (population-size)]

    for gen in range(generation):
        for ind in population:
            indi.evaluate-fitness(3)

        best-ind = select-best-individual(pop, 3)

        new-pop = []
        while len(new-pop) < population-size:
            if random.random() < crossover rate:
                parent1 = random.choice(pop)
                parent2 = random.choice(pop)
                offspring = crossover(parent1, parent2)
                new-pop-append(offspring)
            else:
                individual = random.choice(pop)
                mutate(ind)
                new-pop.append(individual)

        pop = newpop

if __name__ == "__main__":
    run_gp-algo();
```

OUTPUT:

```
Gen 1 : Best-fitness = 0.00070....
Gen 2 :      "       = 0.007716---
Gen 3 :      "       = 0.444444--
Gen 4 :      "       = 1.2
Gen 5 :      "       = 1.333333.....
```

Code:

```
import random
import operator
import math

# Constants for the genetic algorithm
POPULATION_SIZE = 100
GENERATIONS = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
MAX_TREE_DEPTH = 5
FUNCTIONS = ['+', '*', '/']
TERMINALS = ['x', '1', '2', '3']

# Class to represent an individual in the population
class Individual:
    def __init__(self, expression):
        self.expression = expression
        self.fitness = float('inf')

    # Function to evaluate the fitness of an individual
    def evaluate_fitness(self, x_value):
        try:
            expr = self.expression.replace('x', str(x_value))
            # Using eval to evaluate the expression
            self.fitness = eval(expr)
        except Exception as e:
            self.fitness = float('inf')

# Function to generate a random individual
def generate_random_individual():
    expression = generate_random_expression(MAX_TREE_DEPTH)
    return Individual(expression)

# Function to generate a random expression (tree-like structure)
def generate_random_expression(depth):
    if depth == 0 or random.random() < 0.3:
        # Return a terminal (e.g., x or constants)
        return random.choice(TERMINALS)
    else:
        # Return a function with two subexpressions
```

```
        function = random.choice(FUNCTIONS)
        left = generate_random_expression(depth - 1)
        right = generate_random_expression(depth - 1)
        return f"({left} {function} {right})"

# Function to perform crossover between two individuals
def crossover(parent1, parent2):
    # For simplicity, we just swap subexpressions between two individuals
    expr1, expr2 = parent1.expression, parent2.expression
    split1 = random.choice(expr1.split())
    split2 = random.choice(expr2.split())
    offspring_expr = expr1.replace(split1, split2, 1)
    return Individual(offspring_expr)

# Function to mutate an individual
def mutate(individual):
    if random.random() < MUTATION_RATE:
        # Replace a random part of the expression with a new one
        mutated_expr = individual.expression
        split_expr = mutated_expr.split()
        mutated_expr = mutated_expr.replace(random.choice(split_expr),
generate_random_expression(MAX_TREE_DEPTH), 1)
        individual.expression = mutated_expr

# Function to select the best individual
def select_best_individual(population, x_value):
    best_individual = min(population, key=lambda ind: ind.fitness)
    best_individual.evaluate_fitness(x_value)
    return best_individual

# Main function to run the GEP algorithm
def run_gep_algorithm():
    population = [generate_random_individual() for _ in
range(POPULATION_SIZE)]

    for generation in range(GENERATIONS):
        # Evaluate fitness for each individual
        for individual in population:
            individual.evaluate_fitness(3)  # Example with x=3

        # Select the best individual
        best_individual = select_best_individual(population, 3)
```

```
        # Print the fitness of the best individual in each generation
        print(f"Generation {generation + 1}: Best fitness =
{best_individual.fitness}")

        # Create a new population using crossover and mutation
        new_population = []
        while len(new_population) < POPULATION_SIZE:
            if random.random() < CROSSOVER_RATE:
                parent1 = random.choice(population)
                parent2 = random.choice(population)
                offspring = crossover(parent1, parent2)
                new_population.append(offspring)
            else:
                individual = random.choice(population)
                mutate(individual)
                new_population.append(individual)

        population = new_population

# Run the algorithm
if __name__ == "__main__":
    print("Raghavendra R,1BM22CS214")
    run_gep_algorithm()
```

```
# Run the algorithm
if __name__ == "__main__":
    print("Raghavendra R, 1BM22CS214")
    run_gep_algorithm()
```

```
Raghavendra R, 1BM22CS214
Generation 1: Best fitness = 0.007894736842105263
Generation 2: Best fitness = 0.016414141414141416
Generation 3: Best fitness = 0.016414141414141416
Generation 4: Best fitness = 1.0
Generation 5: Best fitness = 1
```