VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Rishi J (1BM22CS222)

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 Rishi J (1BM22CS222), 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. Swathi Sridharan **Assistant Professor** **Department of CSE, BMSCE**	**Dr. Kavitha Sooda** **Professor & HOD** **Department of CSE, BMSCE**

Index

Github Link: **https://github.com/rishibmsce/bislab**

Program 1 - Genetic Algorithms for Optimisation:

Code:

```
import random
def objective_function(x):
    # return max(0, x)
    return abs(x)

def generate_individual(lower_bound, upper_bound):
    return random.uniform(lower_bound, upper_bound)

def create_population(pop_size, lower_bound, upper_bound):
    return [generate_individual(lower_bound, upper_bound) for _ in range(pop_size)]

def fitness(individual):
    return objective_function(individual)

def select(population, fitnesses):
    # total_fitness = sum(fitnesses)
    # print("Current Fitness scores: ", fitnesses)
    # selected = random.choices(population, weights=fitnesses, k=2)
    return [x for _, x in sorted(zip(fitnesses, population), reverse=True)][:2]
    # return selected

def crossover(parent1, parent2):
    alpha = random.random()
    child1 = alpha * parent1 + (1 - alpha) * parent2
    child2 = alpha * parent2 + (1 - alpha) * parent1
    return child1, child2

def mutate(individual, mutation_rate, lower_bound, upper_bound):
    if random.random() < mutation_rate:
        return generate_individual(lower_bound, upper_bound)
    return individual

def genetic_algorithm(pop_size, lower_bound, upper_bound, generations, mutation_rate):
    population = create_population(pop_size, lower_bound, upper_bound)

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

        new_population = []
        for _ in range(pop_size // 2):

            parent1, parent2 = select(population, fitnesses)
```

```
            child1, child2 = crossover(parent1, parent2)

        child1 = mutate(child1, mutation_rate, lower_bound, upper_bound)
        child2 = mutate(child2, mutation_rate, lower_bound, upper_bound)

        new_population.extend([child1, child2])
        population = population[2:]
        # print(population)

      population = new_population

      best_individual = max(population, key=fitness)
      print(f"Generation {generation + 1}: Best = {best_individual}, Fitness =
{fitness(best_individual)}")

   return max(population, key=fitness)

pop_size = 20
lower_bound = -20
upper_bound = 10
generations = 10
mutation_rate = 0.1

best_solution = genetic_algorithm(pop_size, lower_bound, upper_bound, generations, mutation_rate)

print(f"Best solution: {best_solution}, Fitness: {fitness(best_solution)}”)
```

Output:

```
Generation 1: Best = -19.731654314821505, Fitness = 19.731654314821505
Generation 2: Best = -19.718940169213283, Fitness = 19.718940169213283
Generation 3: Best = -19.70895529912996, Fitness = 19.70895529912996
Generation 4: Best = -19.67922032321223, Fitness = 19.67922032321223
Generation 5: Best = -17.499782940547878, Fitness = 17.499782940547878
Generation 6: Best = -19.16331010556096, Fitness = 19.16331010556096
Generation 7: Best = -19.115814136288446, Fitness = 19.115814136288446
Generation 8: Best = -19.114107639819956, Fitness = 19.114107639819956
Generation 9: Best = -19.100566979518746, Fitness = 19.100566979518746
Generation 10: Best = -19.099496648963864, Fitness = 19.099496648963864
```

Program 2 - Particle Swarm Optimisation:

Code:

```
import numpy as np

# Define the objective function to minimize
def objective_function(x):
    return np.sum(x**2 - 10*np.cos(2*np.pi*x) + 10)  # Rastrigin Function (benchmark for 
Optimisation algorithms)

class Particle:
    def __init__(self, dimensions):
        self.position = np.random.uniform(-100, 100, dimensions)  # Initialize positions
        self.velocity = np.random.uniform(-10, 10, dimensions)    # Initialize velocities
        self.best_position = np.copy(self.position)          # Best position of the particle
        self.best_score = objective_function(self.position)     # Best score of the particle

    def update_velocity(self, global_best_position, w=0.729, c1=1.49445, c2=1.49445):
        inertia = w * self.velocity
        cognitive_component = c1 * np.random.random() * (self.best_position - self.position)
        social_component = c2 * np.random.random() * (global_best_position - self.position)
        self.velocity = inertia + cognitive_component + social_component

    def update_position(self):
        self.position += self.velocity

        # Keep particles within bounds (optional, depending on problem)
        self.position = np.clip(self.position, -10, 10)

class ParticleSwarmOptimizer:
    def __init__(self, dimensions, num_particles=30, iterations=100):
        self.dimensions = dimensions
        self.num_particles = num_particles
        self.iterations = iterations
        self.particles = [Particle(dimensions) for _ in range(num_particles)]
        self.global_best_position = np.copy(self.particles[0].position)
        self.global_best_score = objective_function(self.global_best_position)

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

                # Update the personal best of the particle
                if score < particle.best_score:
                    particle.best_score = score
                    particle.best_position = np.copy(particle.position)
```

```python
                # Update the global best position
                if score < self.global_best_score:
                    self.global_best_score = score
                    self.global_best_position = np.copy(particle.position)

            # Update velocity and position of each particle
            for particle in self.particles:
                particle.update_velocity(self.global_best_position)
                particle.update_position()

            print(f"Iteration {iteration + 1}/{self.iterations}, Best Score: {self.global_best_score}")

        return self.global_best_position, self.global_best_score

# Example usage:
if __name__ == "__main__":
    dimensions = 5  # Number of dimensions of the search space
    pso = ParticleSwarmOptimizer(dimensions, num_particles=50, iterations=100)
    best_position, best_score = pso.optimize()

    print("Best position:", best_position)
    print("Best score:", best_score)
```

Output:

```
Iteration 90/100, Best Score: 3.9800784801117235
Iteration 91/100, Best Score: 3.9799937859787295
Iteration 92/100, Best Score: 3.9799494572629825
Iteration 93/100, Best Score: 3.9799263955801436
Iteration 94/100, Best Score: 3.979888176721998
Iteration 95/100, Best Score: 3.9798798709600725
Iteration 96/100, Best Score: 3.9798657092647307
Iteration 97/100, Best Score: 3.9798622618181394
Iteration 98/100, Best Score: 3.9798622618181394
Iteration 99/100, Best Score: 3.9798471887727196
Iteration 100/100, Best Score: 3.9798392125417745
```

Program 3- Ant Colony Optimisation for Travelling Salesman Problem:

Code:

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

# Number of cities
n_cities = 50

# Randomly generate city coordinates
np.random.seed(42)
cities = np.random.rand(n_cities, 2) * 100

# Distance matrix between cities
def distance(c1, c2):
    return np.sqrt((c1[0] - c2[0])**2 + (c1[1] - c2[1])**2)

distance_matrix = np.array([[distance(c1, c2) for c2 in cities] for c1 in cities])

# Ant Colony Optimisation Parameters
n_ants = 20
n_iterations = 100
alpha = 1.0    # Pheromone importance
beta = 1.0     # Distance importance
evaporation_rate = 0.9
Q = 100        # Constant related to pheromone deposition
initial_pheromone = 1.0

# Initialize pheromone matrix
pheromone_matrix = np.ones((n_cities, n_cities)) * initial_pheromone

# Function to choose the next city for an ant based on probability
def select_next_city(current_city, visited, pheromone_matrix, distance_matrix, alpha, beta):
    probabilities = []
    for i in range(n_cities):
        if i not in visited:
            pheromone = pheromone_matrix[current_city][i] ** alpha
            distance = (1 / distance_matrix[current_city][i]) ** beta
            probabilities.append(pheromone * distance)
        else:
            probabilities.append(0)

    probabilities = np.array(probabilities)
    probabilities /= probabilities.sum()
    return np.random.choice(range(n_cities), p=probabilities)

# ACO main loop
```

```
best_route = None
best_length = float('inf')

for iteration in range(n_iterations):
    routes = []
    lengths = []

    # Each ant builds a tour
    for ant in range(n_ants):
        current_city = random.randint(0, n_cities - 1)
        route = [current_city]
        visited = set(route)

        # Build a complete tour
        for _ in range(n_cities - 1):
            next_city = select_next_city(current_city, visited, pheromone_matrix, distance_matrix, alpha,
beta)
            route.append(next_city)
            visited.add(next_city)
            current_city = next_city

        # Complete the tour by returning to the starting city
        route.append(route[0])
        routes.append(route)

        # Calculate the length of the tour
        length = sum([distance_matrix[route[i]][route[i+1]] for i in range(n_cities)])
        lengths.append(length)

        # Update the best route
        if length < best_length:
            best_length = length
            best_route = route

    # Update pheromone levels
    pheromone_matrix *= (1 - evaporation_rate)  # Evaporate pheromone

    # Add pheromone to the routes based on their quality
    for route, length in zip(routes, lengths):
        for i in range(n_cities):
            pheromone_matrix[route[i]][route[i+1]] += Q / length

    # Print progress
    print(f"Iteration {iteration+1}/{n_iterations}, Best Length: {best_length}")

# Plot the best route
def plot_route(cities, route):
    plt.figure(figsize=(10, 5))
    plt.scatter(cities[:, 0], cities[:, 1], c='blue', label='Cities')
```

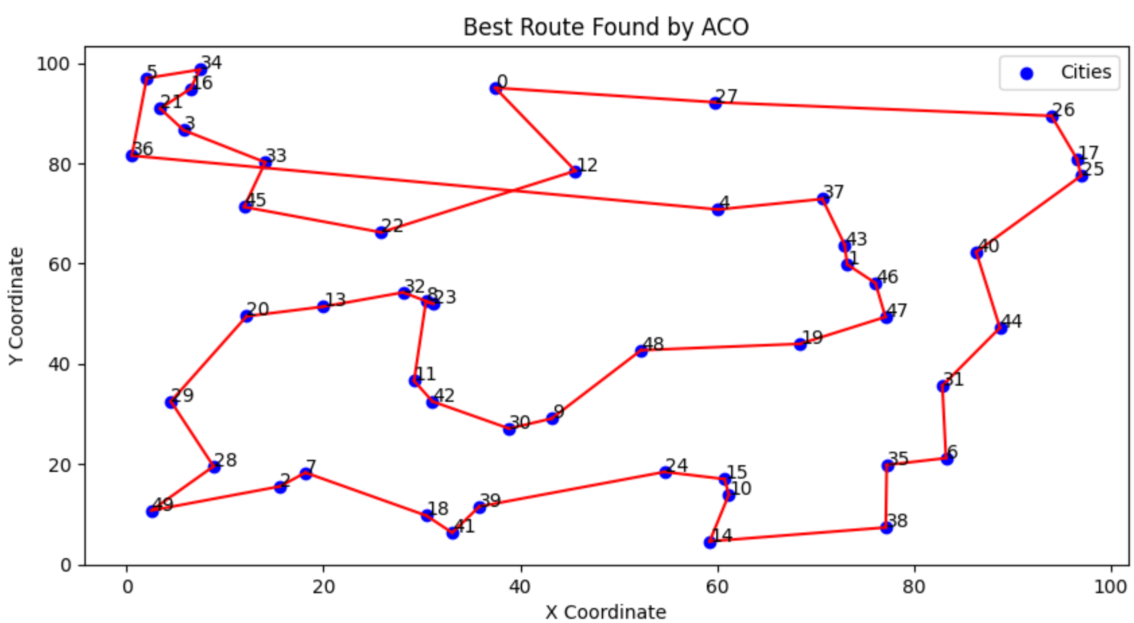
```
    for i, city in enumerate(cities):
        plt.annotate(f'{i}', (city[0], city[1]))

    for i in range(n_cities):
        city1 = cities[route[i]]
        city2 = cities[route[i+1]]
        plt.plot([city1[0], city2[0]], [city1[1], city2[1]], 'r-')

    plt.title('Best Route Found by ACO')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.legend()
    plt.show()

# Display the best route
print("Best route:", best_route)
plot_route(cities, best_route)
```

Output:




```
Iteration 91/100, Best Length: 665.2149872633906
Iteration 92/100, Best Length: 665.2149872633906
Iteration 93/100, Best Length: 665.2149872633906
Iteration 94/100, Best Length: 665.2149872633906
Iteration 95/100, Best Length: 665.2149872633906
Iteration 96/100, Best Length: 665.2149872633906
Iteration 97/100, Best Length: 665.2149872633906
Iteration 98/100, Best Length: 665.2149872633906
Iteration 99/100, Best Length: 665.2149872633906
Iteration 100/100, Best Length: 665.2149872633906
```

Program 4 - Cuckoo Search Optimisation:

Code:

```
import numpy as np

def objective_function(x):
    # Example objective function: Sphere function
    return np.sum(x**2)

def levy_flight(Lambda):
    # Generate a step size following a Lévy distribution
    sigma_u = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
               (np.math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.randn() * sigma_u
    v = np.random.randn()
    step = u / abs(v) ** (1 / Lambda)
    return step

def cuckoo_search(num_nests=25, max_iter=100, pa=0.25, lb=-5, ub=5, dim=2):
    # Initialize nests
    nests = np.random.uniform(lb, ub, (num_nests, dim))
    fitness = np.array([objective_function(n) for n in nests])

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

    for iteration in range(max_iter):
        new_nests = np.copy(nests)

        # Lévy flights and update nests
        for i in range(num_nests):
            step_size = levy_flight(1.5)
            step = step_size * (nests[i] - best_nest)
            new_nests[i] = nests[i] + step * np.random.randn(dim)
            new_nests[i] = np.clip(new_nests[i], lb, ub)

        # Evaluate fitness for new nests
        new_fitness = np.array([objective_function(n) for n in new_nests])

        # Replace some of the nests based on the fitness
        for i in range(num_nests):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        # Sort and update the best nest
        min_fitness_index = np.argmin(fitness)
        if fitness[min_fitness_index] < best_fitness:
```

```python
            best_nest = nests[min_fitness_index]
            best_fitness = fitness[min_fitness_index]

        # Abandon some nests with probability pa and create new nests
        for i in range(num_nests):
            if np.random.rand() < pa:
                nests[i] = np.random.uniform(lb, ub, dim)
                fitness[i] = objective_function(nests[i])

        # Display the best fitness at each iteration
        print(f"Iteration {iteration+1}/{max_iter}, Best Fitness: {best_fitness}")

    return best_nest, best_fitness

# Usage
best_solution, best_value = cuckoo_search()
print(f"best solution: {best_solution}, best value: {best_value}")
```

Output:

```
Iteration 90/100, Best Fitness: 0.028302717555542513
Iteration 91/100, Best Fitness: 0.028302717555542513
Iteration 92/100, Best Fitness: 0.028302717555542513
Iteration 93/100, Best Fitness: 0.028302717555542513
Iteration 94/100, Best Fitness: 0.028302717555542513
Iteration 95/100, Best Fitness: 0.028302717555542513
Iteration 96/100, Best Fitness: 0.028302717555542513
Iteration 97/100, Best Fitness: 0.028302717555542513
Iteration 98/100, Best Fitness: 0.028302717555542513
Iteration 99/100, Best Fitness: 0.028302717555542513
Iteration 100/100, Best Fitness: 0.028302717555542513
```

Program 5 - Grey Wolf Optimisation:

Code:

```
import numpy as np

# Define the objective function (e.g., Sphere function)
def objective_function(x):
    return np.sum(x**2 - 10*np.cos(2*np.pi*x) + 10)

# Grey Wolf Optimizer
class GreyWolfOptimizer:
    def __init__(self, obj_func, dim, n_wolves=20, max_iter=100, lb=-10, ub=10):
        self.obj_func = obj_func
        self.dim = dim  # Dimension of the problem
        self.n_wolves = n_wolves  # Number of wolves (population size)
        self.max_iter = max_iter  # Maximum number of iterations
        self.lb = lb  # Lower bound
        self.ub = ub  # Upper bound

        # Initialize the wolves randomly
        self.positions = np.random.uniform(lb, ub, (n_wolves, dim))
        self.alpha_pos = np.zeros(dim)
        self.beta_pos = np.zeros(dim)
        self.delta_pos = np.zeros(dim)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")
        self.delta_score = float("inf")

    def optimize(self):
        for t in range(self.max_iter):
            # Update the positions of alpha, beta, and delta
            for i in range(self.n_wolves):
                fitness = self.obj_func(self.positions[i])

                # Update alpha, beta, delta based on fitness
                if fitness < self.alpha_score:
                    self.delta_score = self.beta_score
                    self.delta_pos = self.beta_pos.copy()
                    self.beta_score = self.alpha_score
                    self.beta_pos = self.alpha_pos.copy()
                    self.alpha_score = fitness
                    self.alpha_pos = self.positions[i].copy()
                elif fitness < self.beta_score:
                    self.delta_score = self.beta_score
                    self.delta_pos = self.beta_pos.copy()
                    self.beta_score = fitness
                    self.beta_pos = self.positions[i].copy()
                elif fitness < self.delta_score:
```

```python
                    self.delta_score = fitness
                    self.delta_pos = self.positions[i].copy()

            # Update each wolf's position
            a = 2 - t * (2 / self.max_iter)  # Linearly decreases from 2 to 0
            for i in range(self.n_wolves):
                for j in range(self.dim):
                    r1, r2 = np.random.rand(), np.random.rand()
                    A1 = 2 * a * r1 - a
                    C1 = 2 * r2
                    D_alpha = abs(C1 * self.alpha_pos[j] - self.positions[i, j])
                    X1 = self.alpha_pos[j] - A1 * D_alpha

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

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

                    # Update wolf position
                    self.positions[i, j] = (X1 + X2 + X3) / 3

                # Boundary check
                self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)

        return self.alpha_score, self.alpha_pos

# Example usage
dim = 5  # Number of dimensions
optimizer = GreyWolfOptimizer(objective_function, dim=dim, n_wolves=20, max_iter=100, lb=-10, ub=10)
best_score, best_position = optimizer.optimize()
print(f"best score: {best_score}; best position: {best_position}")
```

Output:

```
best score: 5.3624624351056775; best position: [-0.07973623  0.02956324  1.06223069  0.01585228 -0.10270275]
```

Program 6 - Parallel Cellular Algorithms:

Code:

```
import numpy as np
import random

# Step 1: Define the Problem (Optimisation 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]])
```

```python
    # 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

    new_position = np.clip(new_position, minx, maxx)
    return new_position
population = initialize_population(grid_size, dim, minx, maxx)
for iteration in range(max_iterations):
    fitness_grid = evaluate_fitness(population)

    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

    best_fitness = np.min(fitness_grid)
    print(f"Iteration {iteration + 1}, Best Fitness: {best_fitness}")

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("Best Position Found:", best_position)
print("Best Fitness Found:", best_fitness)
```

Output:

```
Iteration 40, Best Fitness: 0.0003142924839317396
Iteration 41, Best Fitness: 0.00017229857179437203
Iteration 42, Best Fitness: 0.0003510862276322362
Iteration 43, Best Fitness: 3.138096594710152e-05
Iteration 44, Best Fitness: 0.00015400826922191437
Iteration 45, Best Fitness: 2.925206121864419e-05
Iteration 46, Best Fitness: 0.00015368193895130472
Iteration 47, Best Fitness: 0.00018335840951939198
Iteration 48, Best Fitness: 1.2570637668601874e-06
Iteration 49, Best Fitness: 5.413914877609205e-05
Iteration 50, Best Fitness: 1.4336632028623929e-06
```

Program 7 - Gene Expression Algorithms:

Code:

```
import random
import numpy as np

def fitness_function(x):
    return np.sum(x**2 - 10 * np.cos(2 * np.pi * x) + 10)  # Negative to maximize

def create_population(population_size, gene_length):
    population = []
    for _ in range(population_size):
        chromosome = [random.uniform(-5, 5) for _ in range(gene_length)]
        population.append(chromosome)
    return population

def selection(population, fitness_scores):
    selected_parents = []
    for _ in range(2):
        max_fitness = max(fitness_scores)
        max_index = fitness_scores.index(max_fitness)
        selected_parents.append(population[max_index])
        fitness_scores[max_index] = -float("inf")  # Prevent repeated selection
    return selected_parents

def crossover(parent1, parent2, crossover_rate):
    child1, child2 = parent1.copy(), parent2.copy()
    if random.random() < crossover_rate:
        crossover_point = random.randint(0, len(parent1) - 1)
        child1[crossover_point:], child2[crossover_point:] = (
            parent2[crossover_point:],
            parent1[crossover_point:],
        )
    return child1, child2

def mutation(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += random.uniform(-0.5, 0.5)
    return individual


def genetic_algorithm(
    population_size, gene_length, num_generations, crossover_rate, mutation_rate
):
    population = create_population(population_size, gene_length)
    for generation in range(num_generations):
```

```
        fitness_scores = [fitness_function(individual[0]) for individual in population]
        best_fitness = max(fitness_scores)
        best_x = population[fitness_scores.index(best_fitness)][0]
        print(
            f"Generation {generation+1}: Best fitness = {best_fitness:.4f}, Best x = {best_x:.4f}"
        )
        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = selection(population, fitness_scores.copy())
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            child1 = mutation(child1, mutation_rate)
            child2 = mutation(child2, mutation_rate)
            new_population.extend([child1, child2])
        population = new_population
    best_individual = max(population, key=lambda x: fitness_function(x[0]))
    return best_individual[0]


# Example usage
best_x = genetic_algorithm(
    population_size=100,
    gene_length=1,
    num_generations=100,
    crossover_rate=0.8,
    mutation_rate=0.1,
)
print("Final Solution: Best x =", best_x)
print("Final Solution: Best fitness =", fitness_function(best_x))
```

Output:

```
Generation 90: Best fitness = 40.3533, Best x = 4.5234
Generation 91: Best fitness = 40.3533, Best x = 4.5234
Generation 92: Best fitness = 40.3533, Best x = 4.5234
Generation 93: Best fitness = 40.3533, Best x = 4.5234
Generation 94: Best fitness = 40.3533, Best x = 4.5234
Generation 95: Best fitness = 40.3533, Best x = 4.5234
Generation 96: Best fitness = 40.3533, Best x = 4.5234
Generation 97: Best fitness = 40.3533, Best x = 4.5234
Generation 98: Best fitness = 40.3533, Best x = 4.5234
Generation 99: Best fitness = 40.3533, Best x = 4.5234
Generation 100: Best fitness = 40.3533, Best x = 4.5234
```