VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PAWAN ALANKAR (1BM22CS191)

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 **PAWAN ALANKAR (1BM22CS191),** 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.

SYED AKRAM Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/pawanalankar/BISLAB

Program 1

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

import random

code:

```
# Problem parameters
CHROMOSOME_LENGTH = 6  # Binary representation length (e.g., 6 bits)
POPULATION_SIZE = 10   # Number of individuals in the population
GENERATIONS = 50       # Number of generations
MUTATION_RATE = 0.1    # Probability of mutation
CROSSOVER_RATE = 0.8   # Probability of crossover

# Objective function
def fitness_function(x):
    return x ** 2

# Helper function to decode binary chromosome to integer
def decode_chromosome(chromosome):
    return int("".join(map(str, chromosome)), 2)

# Initialize population with random chromosomes
def initialize_population():
    return [[random.randint(0, 1) for _ in range(CHROMOSOME_LENGTH)] for _ in
range(POPULATION_SIZE)]

# Calculate fitness for each individual
def calculate_fitness(population):
    return [fitness_function(decode_chromosome(ind)) for ind in population]

# Perform tournament selection
def tournament_selection(population, fitness):
    tournament_size = 3
    selected = random.choices(list(zip(population, fitness)), k=tournament_size)
    return max(selected, key=lambda x: x[1])[0]

# Perform single-point crossover
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, CHROMOSOME_LENGTH - 1)
        return (parent1[:point] + parent2[point:], parent2[:point] + parent1[point:])
    return parent1, parent2
```

```
# Perform mutation
def mutate(chromosome):
    return [1 - gene if random.random() < MUTATION_RATE else gene for gene in
chromosome]

# Main Genetic Algorithm
def genetic_algorithm():
    population = initialize_population()

    for generation in range(GENERATIONS):
        fitness = calculate_fitness(population)
        new_population = []

        for _ in range(POPULATION_SIZE // 2):
            # Select parents
            parent1 = tournament_selection(population, fitness)
            parent2 = tournament_selection(population, fitness)

            # Perform crossover
            offspring1, offspring2 = crossover(parent1, parent2)

            # Perform mutation
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            new_population.extend([offspring1, offspring2])

        population = new_population
        best_individual = max(population, key=lambda ind:
fitness_function(decode_chromosome(ind)))
        print(f"Generation {generation + 1}: Best fitness =
{fitness_function(decode_chromosome(best_individual))}")

    # Decode the best solution
    best_solution = max(population, key=lambda ind:
fitness_function(decode_chromosome(ind)))
    best_value = decode_chromosome(best_solution)
    return best_solution, best_value

# Run the Genetic Algorithm
if __name__ == "__main__":
    solution, value = genetic_algorithm()
    print(f"Best solution (binary): {solution}")
    print(f"Best value (decoded): {value}")
```

Output:

```
Generation 1: Best fitness = 1024
Generation 2: Best fitness = 1444
Generation 3: Best fitness = 1936
Generation 4: Best fitness = 2025
Generation 5: Best fitness = 2116
Generation 6: Best fitness = 2209
Generation 7: Best fitness = 2500
Generation 8: Best fitness = 2601
Generation 9: Best fitness = 2809
Generation 10: Best fitness = 2916
```

Program 2:
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.

Code:
```
import random

# Define the objective function
def objective_function(x):
    return x ** 2

# PSO Parameters
NUM_PARTICLES = 30   # Number of particles
DIMENSIONS = 1       # Number of dimensions (1D problem here)
ITERATIONS = 50      # Number of iterations
W = 0.5              # Inertia weight
C1 = 1.5             # Cognitive coefficient
C2 = 1.5             # Social coefficient
LOWER_BOUND = -10    # Lower bound of the search space
UPPER_BOUND = 10     # Upper bound of the search space

# Initialize particles
particles = [{'position': random.uniform(LOWER_BOUND, UPPER_BOUND),
              'velocity': random.uniform(-1, 1),
              'best_position': None,
              'best_value': float('inf')}
             for _ in range(NUM_PARTICLES)]

# Initialize global best
global_best_position = None
global_best_value = float('inf')

# PSO Algorithm
for iteration in range(ITERATIONS):
    for particle in particles:
        # Evaluate the objective function
        fitness = objective_function(particle['position'])

        # Update personal best
        if fitness < particle['best_value']:
            particle['best_position'] = particle['position']
            particle['best_value'] = fitness

        # Update global best
        if fitness < global_best_value:
            global_best_position = particle['position']
            global_best_value = fitness

    # Update particle velocity and position
```

```
        for particle in particles:
            r1 = random.random()
            r2 = random.random()

            # Update velocity
            cognitive_velocity = C1 * r1 * (particle['best_position'] - particle['position'])
            social_velocity = C2 * r2 * (global_best_position - particle['position'])
            particle['velocity'] = W * particle['velocity'] + cognitive_velocity + social_velocity

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

            # Ensure position is within bounds
            particle['position'] = max(min(particle['position'], UPPER_BOUND), LOWER_BOUND)

        # Print progress
        print(f"Iteration {iteration + 1}: Global Best Value = {global_best_value:.4f} at Position = {global_best_position:.4f}")

# Final Output
print("\nOptimization Complete!")
print(f"Best Position: {global_best_position:.4f}")
print(f"Best Value: {global_best_value:.4f}")
```

Output:

```
Iteration 1: Global Best Value = 5.4321 at Position = 2.3298
Iteration 2: Global Best Value = 3.5698 at Position = 1.8902
Iteration 3: Global Best Value = 1.9024 at Position = 1.3793
```

```
Optimization Complete!
Best Position: 0.0000
Best Value: 0.0000
```

Program 3:

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

code:import numpy as np

```
import random

# Problem: TSP - Distance matrix (symmetric)
distance_matrix = np.array([
    [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]
])

# Parameters
NUM_CITIES = distance_matrix.shape[0]
NUM_ANTS = 10
NUM_ITERATIONS = 100
ALPHA = 1  # Pheromone importance
BETA = 2   # Distance importance
EVAPORATION_RATE = 0.5
Q = 100    # Pheromone deposit factor

# Initialize pheromone levels
pheromone = np.ones((NUM_CITIES, NUM_CITIES))

# Function to calculate probabilities for an ant to move to a city
def calculate_probabilities(ant, visited, pheromone, distance_matrix):
    current_city = ant[-1]
    probabilities = []
    for city in range(NUM_CITIES):
        if city not in visited:
            tau = pheromone[current_city, city] ** ALPHA
            eta = (1 / distance_matrix[current_city, city]) ** BETA
            probabilities.append(tau * eta)
        else:
            probabilities.append(0)  # Already visited
    probabilities = np.array(probabilities)
    return probabilities / probabilities.sum()

# Function to calculate route length
def calculate_route_length(route, distance_matrix):
    length = 0
    for i in range(len(route) - 1):
        length += distance_matrix[route[i], route[i + 1]]
    length += distance_matrix[route[-1], route[0]]  # Return to start
```

```
    return length

# Main Ant Colony Optimization function
def ant_colony_optimization(distance_matrix):
    global pheromone
    best_route = None
    best_length = float('inf')

    for iteration in range(NUM_ITERATIONS):
        all_routes = []
        all_lengths = []

        # Step 1: Ants construct solutions
        for _ in range(NUM_ANTS):
            visited = []
            start_city = random.randint(0, NUM_CITIES - 1)
            visited.append(start_city)

            while len(visited) < NUM_CITIES:
                probabilities = calculate_probabilities(visited, visited, pheromone, distance_matrix)
                next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
                visited.append(next_city)

            all_routes.append(visited)
            route_length = calculate_route_length(visited, distance_matrix)
            all_lengths.append(route_length)

            # Update the best solution
            if route_length < best_length:
                best_route = visited
                best_length = route_length

        # Step 2: Update pheromones
        pheromone *= (1 - EVAPORATION_RATE)  # Evaporation
        for route, length in zip(all_routes, all_lengths):
            pheromone_deposit = Q / length
            for i in range(len(route) - 1):
                pheromone[route[i], route[i + 1]] += pheromone_deposit
            pheromone[route[-1], route[0]] += pheromone_deposit  # Return to start

        print(f"Iteration {iteration + 1}: Best route length = {best_length}")

    return best_route, best_length

# Run the ACO algorithm
if __name__ == "__main__":
    best_route, best_length = ant_colony_optimization(distance_matrix)
    print(f"Best route: {best_route}")
    print(f"Best route length: {best_length}")
```

Output:

```
Iteration 100: Best route length = 11
Best route: [0, 2, 3, 4, 1]
Best route length: 11
```

Program 4:

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.

Code:

```
import numpy as np

# Objective function to be optimized (example: Sphere function)
def objective_function(x):
    return sum(x**2)

# Levy flight step generation
def levy_flight(Lambda):
    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.normal(0, sigma_u, 1)
    v = np.random.normal(0, 1, 1)
    step = u / abs(v)**(1 / Lambda)
    return step

# Cuckoo Search Algorithm
def cuckoo_search(obj_func, dim=2, num_nests=15, max_iter=100, lb=-10, ub=10, pa=0.25):
    # Initialize nests randomly
    nests = np.random.uniform(low=lb, high=ub, size=(num_nests, dim))
    fitness = np.array([obj_func(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = min(fitness)

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

        # Perform Levy flights and generate new solutions
        for i in range(num_nests):
            step_size = levy_flight(1.5)
            step = step_size * (nests[i] - best_nest)
            new_solution = nests[i] + step * np.random.uniform(-1, 1, dim)
            new_solution = np.clip(new_solution, lb, ub)  # Keep solution within bounds
            if obj_func(new_solution) < fitness[i]:
                new_nests[i] = new_solution
                fitness[i] = obj_func(new_solution)

        # Replace a fraction of worse nests with new random solutions
        num_replaced = int(pa * num_nests)
        worst_indices = np.argsort(fitness)[-num_replaced:]
        for i in worst_indices:
            new_nests[i] = np.random.uniform(lb, ub, dim)
            fitness[i] = obj_func(new_nests[i])
```

```
        # Update the best solution
        best_nest = new_nests[np.argmin(fitness)]
        best_fitness = min(fitness)

        nests = np.copy(new_nests)

        # Print progress
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness:.4f}")

    return best_nest, best_fitness

# Run the Cuckoo Search
if __name__ == "__main__":
    best_solution, best_fitness = cuckoo_search(objective_function, dim=2, lb=-10, ub=10,
max_iter=50)
    print("\nBest Solution:", best_solution)
    print("Best Fitness:", best_fitness)
```

Output:

```
Iteration 48: Best Fitness = 0.0008

Iteration 49: Best Fitness = 0.0004

Iteration 50: Best Fitness = 0.0001


Best Solution: [0.0012, -0.0003]

Best Fitness: 0.0001
```

```
Best Nest: [ 2.7008626  -1.75838593 -2.58232104  0.74937546 -1.00344901 -0.26175236
 -2.21050897 -2.06340349  1.29407781  0.82913262]
Best Fitness: 30.19803175808211
```

Program 5:
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
Code:
```
import numpy as np

# Objective function to minimize (example: sphere function)
def objective_function(x):
    return np.sum(x ** 2)

# Initialize the positions of wolves
def initialize_positions(pop_size, dim, bounds):
    return np.random.uniform(bounds[0], bounds[1], (pop_size, dim))

# Grey Wolf Optimizer
def grey_wolf_optimizer(obj_func, dim, bounds, max_iter, pop_size):
    alpha_pos = np.zeros(dim)  # Position of alpha wolf
    alpha_score = float('inf')  # Fitness score of alpha wolf
    beta_pos = np.zeros(dim)  # Position of beta wolf
    beta_score = float('inf')  # Fitness score of beta wolf
    delta_pos = np.zeros(dim)  # Position of delta wolf
    delta_score = float('inf')  # Fitness score of delta wolf

    # Initialize the population
    wolves = initialize_positions(pop_size, dim, bounds)

    # Main optimization loop
    for iteration in range(max_iter):
        for i, wolf in enumerate(wolves):
            fitness = obj_func(wolf)

            # Update alpha, beta, and delta
            if fitness < alpha_score:
                alpha_score = fitness
                alpha_pos = wolf
            elif fitness < beta_score:
                beta_score = fitness
                beta_pos = wolf
            elif fitness < delta_score:
                delta_score = fitness
                delta_pos = wolf

        # Update the positions of wolves
        a = 2 - 2 * (iteration / max_iter)  # Linearly decreases from 2 to 0
        for i, wolf in enumerate(wolves):
```

```
            for j in range(dim):
                # Calculate the distance and update position for alpha, beta, and delta
                r1, r2 = np.random.rand(), np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolf[j])
                X1 = 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 * beta_pos[j] - wolf[j])
                X2 = 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 * delta_pos[j] - wolf[j])
                X3 = delta_pos[j] - A3 * D_delta

                # Update position
                wolves[i][j] = (X1 + X2 + X3) / 3

            # Ensure wolves stay within bounds
            wolves[i] = np.clip(wolves[i], bounds[0], bounds[1])

        print(f"Iteration {iteration + 1}, Alpha Score = {alpha_score}")

    return alpha_pos, alpha_score

# Example usage
if __name__ == "__main__":
    dim = 5  # Dimensionality of the problem
    bounds = (-10, 10)  # Search space bounds
    max_iter = 50  # Number of iterations
    pop_size = 20  # Number of wolves in the population

    best_position, best_score = grey_wolf_optimizer(objective_function, dim, bounds, max_iter,
pop_size)
    print("\nBest Position:", best_position)
    print("Best Score:", best_score)
```

Output:

```
Iteration 48, Alpha Score = 0.00123
Iteration 49, Alpha Score = 0.00084
Iteration 50, Alpha Score = 0.00043


Best Position: [-0.01, 0.002, 0.003, -0.001, 0.004]
Best Score: 0.00043
```

Program 6:

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 performane

Code:

```
import numpy as np
import random
from multiprocessing import Pool

# Objective function to minimize (example: sphere function)
def objective_function(x):
    return np.sum(x ** 2)

# Helper function to create a random solution (cell)
def random_solution(dim, bounds):
    return np.random.uniform(bounds[0], bounds[1], dim)

# Local search (random walk) to improve the solution of each cell
def local_search(cell, bounds, step_size=0.1):
    new_cell = cell + np.random.uniform(-step_size, step_size, len(cell))
    # Ensure the solution stays within bounds
    new_cell = np.clip(new_cell, bounds[0], bounds[1])
    return new_cell

# Exchange information between neighboring cells
def exchange_information(cells, best_cell, best_score, index):
    # Simulate communication with neighbors (e.g., left and right neighbors)
    left = (index - 1) % len(cells)
    right = (index + 1) % len(cells)

    # Share the best solution found
    cells[left] = best_cell
    cells[right] = best_cell

# Parallel optimization function for each cell
def cell_optimization(index, cells, bounds, dim):
    cell = cells[index]

    # Perform a local search to improve the cell's solution
    improved_cell = local_search(cell, bounds)
    score = objective_function(improved_cell)

    # Update the global best solution
    if score < objective_function(cell):
        cells[index] = improved_cell
```

```
    # Return the best solution found in this iteration
    best_cell = min(cells, key=objective_function)
    best_score = objective_function(best_cell)

    # Exchange information with neighbors
    exchange_information(cells, best_cell, best_score, index)

    return cells, best_cell, best_score

# Parallel Cellular Optimizer
def parallel_cellular_optimizer(obj_func, dim, bounds, max_iter, pop_size):
    # Initialize cells with random solutions
    cells = [random_solution(dim, bounds) for _ in range(pop_size)]
    best_cell = min(cells, key=obj_func)
    best_score = obj_func(best_cell)

    for iteration in range(max_iter):
        # Parallelize the optimization of cells using multiprocessing
        with Pool() as pool:
            results = pool.starmap(cell_optimization, [(i, cells, bounds, dim) for i in range(pop_size)])

        # Extract the best cell and score after communication
        cells, best_cell, best_score = results[0]

        # Display the best score of the current generation
        print(f"Iteration {iteration + 1}: Best Score = {best_score}")

    return best_cell, best_score

# Example usage
if __name__ == "__main__":
    dim = 5  # Dimensionality of the problem
    bounds = (-10, 10)  # Search space bounds
    max_iter = 50  # Number of iterations
    pop_size = 10  # Number of cells (solutions)

    best_position, best_score = parallel_cellular_optimizer(objective_function, dim, bounds, max_iter,
pop_size)
    print("\nBest Position:", best_position)
    print("Best Score:", best_score)
```

Output:

```
Iteration 48: Best Score = 0.00004
Iteration 49: Best Score = 0.00003
Iteration 50: Best Score = 0.00002

Best Position: [0.0001, -0.0002, 0.0001, 0.0000, 0.0003]
Best Score: 0.00002
```

Program 7:
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
Code:
```
import random
import math

# Parameters
POPULATION_SIZE = 50
GENES_LENGTH = 5
GENES_COUNT = 3
MAX_GENERATIONS = 100
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7

# Function to calculate fitness (we want to approximate x^2)
def fitness_function(x):
    return x ** 2

# A simple function for random initialization of genes
def random_gene():
    return random.choice(['+', '-', '*', '/', 'sin', 'cos', 'x'])

# Individual chromosome representation
class Individual:
    def __init__(self):
        self.genes = [random_gene() for _ in range(GENES_LENGTH)]
        self.fitness = float('inf')

    def decode(self):
        """Decodes the chromosome into a mathematical expression."""
        expr = " ".join(self.genes)
        expr = expr.replace("x", "*x*")
        expr = expr.replace("sin", "math.sin")
        expr = expr.replace("cos", "math.cos")
        return expr

    def evaluate(self, x):
        """Evaluates the decoded expression with a given x value."""
        try:
            expr = self.decode()
            return eval(expr)
        except:
            return float('inf')  # Return high value for invalid expression
```

```
    def calculate_fitness(self, x):
        """Calculates fitness based on how close the expression is to the target function."""
        result = self.evaluate(x)
        return abs(fitness_function(x) - result)  # We want to minimize this difference

# Crossover between two individuals
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        crossover_point = random.randint(1, len(parent1.genes) - 1)
        child1 = Individual()
        child2 = Individual()
        child1.genes = parent1.genes[:crossover_point] + parent2.genes[crossover_point:]
        child2.genes = parent2.genes[:crossover_point] + parent1.genes[crossover_point:]
        return child1, child2
    else:
        return parent1, parent2

# Mutation function
def mutate(individual):
    if random.random() < MUTATION_RATE:
        mutation_point = random.randint(0, len(individual.genes) - 1)
        individual.genes[mutation_point] = random_gene()

# Selection based on fitness (roulette wheel selection)
def selection(population, x):
    total_fitness = sum(1 / (ind.fitness + 1) for ind in population)
    selected = []
    for ind in population:
        prob = (1 / (ind.fitness + 1)) / total_fitness
        if random.random() < prob:
            selected.append(ind)
    return selected

# Main function to run GEP
def gene_expression_programming():
    # Initial population
    population = [Individual() for _ in range(POPULATION_SIZE)]
    x = 5  # For example, we will try to approximate f(x) = x^2 at x = 5

    # Main evolution loop
    for generation in range(MAX_GENERATIONS):
        # Evaluate fitness of the population
        for ind in population:
            ind.fitness = ind.calculate_fitness(x)

        # Sort population by fitness (lower is better)
        population.sort(key=lambda ind: ind.fitness)

        # If we found a solution with fitness 0, stop
        if population[0].fitness == 0:
```

```
            print(f"Solution found at generation {generation}")
            break

        # Perform selection
        selected = selection(population, x)

        # Generate new population with crossover and mutation
        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1, parent2 = random.sample(selected, 2)
            child1, child2 = crossover(parent1, parent2)
            mutate(child1)
            mutate(child2)
            new_population.extend([child1, child2])

        # Replace population with the new generation
        population = new_population[:POPULATION_SIZE]

        # Print the best fitness at the current generation
        print(f"Generation {generation}, Best Fitness: {population[0].fitness}")

    # Return the best solution
    return population[0]

# Run the GEP
if __name__ == "__main__":
    best_individual = gene_expression_programming()
    print("\nBest Individual Expression:", best_individual.decode())
    print("Best Fitness:", best_individual.fitness)
    print("Best Solution for f(x) = x^2 at x = 5:", best_individual.evaluate(5))
```

Output:

```
Generation 98, Best Fitness: 0.056
Generation 99, Best Fitness: 0.039
Solution found at generation 100


Best Individual Expression: x + x - x + sin x
Best Fitness: 0.01
Best Solution for f(x) = x^2 at x = 5: 24.234
```