Vf5

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreyas T S (1BM23CS322)

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
Aug-2025 to Jan-2026

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 **Shreyas T S (1BM23CS322),** 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.

Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Shreyes45/BIS-lab

Program 1

Genetic Algorithm for Optimization Problems

Lab - 1

Genetic Algorithm : 5 main phases
- Initialization
- Fitness Assignment
- Selection
- Crossover
- Termination

Iteration - 1

Steps

1) Selecting encoding technique
0 to 31

2) Select the initial population - '4'

Sl No	Initial population	x value	fitness $f(x)=x^2$	prob $f(x)/\Sigma f(x)$	% Prob	Expected count $f(x)/Avg(f(x))$	Actual count (After, value)
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.084	0
4	10011	19	361	0.3125	31.25	1.25	1

3) Select Mating pool

Sl No	mating pool	crossover point	offspring after crossover	x value	fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289

4) Crossover : Random 4 & 2

5) Mutation

Sl no	offspring after crossover	mutation chromosome for flipping	offspring after mutation	x value	fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400

Pseudo code

```
function fitness(x):
    Return x^2

Function generate_population(pop_size, lower_bound, upper_bound):
    population = []
    for i from 1 to pop_size:
        individual = RandomNumberInRange(lower_bound, upper_bound)
        Add individual to population
    return population

function select_parents(population):
    fitness_values = []
    for each individual in population:
        fitness_values.append(fitness(individual))
    total_fitness = sum(fitness_values)

    selection_probs = []
    for each fitness_value in fitness_values:
        selection_probs.append(fitness_value / total_fitness)
    parent1, parent2 = RandomlySelectTwoIndividualsBasedOnSelectionProbabilities(population, selection_probs)
    return parent1, parent2

Function crossover(parent1, parent2):
    crossover_point = RandomNumberInRange(0, 1)
    child1 = crossover_point * parent1 + (1 - crossover_point) * parent2
    child2 = crossover_point * parent2 + (1 - crossover_point) * parent1
    return child1, child2

Function mutate(individual, mutation_rate, lower_bound, upper_bound):
    if Random() < mutation_rate:
        mutation = RandomNumberInRange(lower_bound, upper_bound)
        individual = individual + mutation
        individual = clamp(individual, lower_bound, upper_bound)
    return individual

Function genetic_algorithm(pop_size, generations, lower_bound, upper_bound, mutation_rate):
    population = generate_population(pop_size, lower_bound, upper_bound)
    for generation from 1 to generations:
        new_population = []
        for i from 1 to pop_size/2:
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate, lower_bound, upper_bound)
            child2 = mutate(child2, mutation_rate, lower_bound, upper_bound)
            Add child1 and child2 to new_population

        population = new_population
        best_individual = findBestIndividual(population)
        print "Generation", generation, ": Best fitness - ", fitness(best_individual), "Best Solution :", best_individual
    best_solution = findBestIndividual(population)
    return best_solution

// Main program
Define population_size = 100
Define generation = 5
Define lower_bound = 0
Define upper_bound = 31
Define mutation_rate = 0.1
best_solution = genetic_algorithm(population_size, lower_bound, upper_bound, mutation_rate)
print "Best Solution found:", best_solution, "Fitness", fitness(best_solution)
```

Assistant Professor
Department of CSE, BMSCE

Professor & HOD
Department of CSE,

output:

Generation 1 : Best fitness = 961 , Best Solution 31
Generation 2 : Best fitness = 961 , Best Solution 31
Generation 3 : Best fitness = 877.017600833, Best Solution : 29.61758040
Generation 4 : Best fitness = 877.81760833368, Best Solution = 29.6138561449
Generation 5 : Best fitness = 961 , Best Solution = 31
Best Solution found : 31 , fitness : 961

applications :
* Travelling salesman problem
* Knapsack problem
* function optimization

Code:

```
import random

def fitness_function(x):
    return x ** 2

def decode(chromosome):
    return int(chromosome, 2)

def evaluate_population(population):
    return [fitness_function(decode(individual)) for individual in population]

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    if total_fitness == 0:
        return random.choice(population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual

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

def mutate(chromosome):
    new_chromosome = ''
    for bit in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome += '0' if bit == '1' else '1'
        else:
            new_chromosome += bit
    return new_chromosome

def get_initial_population(size, length):
    population = []
    print(f"Enter {size} chromosomes (each of {length} bits, e.g., '10101'):")
    while len(population) < size:
        chrom = input(f"Chromosome {len(population)+1}: ").strip()
        if len(chrom) == length and all(bit in '01' for bit in chrom):
            population.append(chrom)
        else:
            print(f"Invalid input. Please enter a {length}-bit binary string.")
    return population
```

```python
def genetic_algorithm():
    population = get_initial_population(POPULATION_SIZE, CHROMOSOME_LENGTH)
    best_solution = None
    best_fitness = float('-inf')

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        for i, individual in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = individual

        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best x =
{decode(best_solution)}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

    print("\nBest solution found:")
    print(f"Chromosome: {best_solution}")
    print(f"x = {decode(best_solution)}")
    print(f"f(x) = {fitness_function(decode(best_solution))}")

POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.8
GENERATIONS = 20

if __name__ == "__main__":
    genetic_algorithm()
```

Program 2

Particle Swarm Optimization for Function Optimization

Lab-2

Particle Swarm optimization

Pseudo Code

1) P = particle initialization();
2) for i = 1 to max
   3) for each particle p in P do
      $f_p = f(p)$
4) if f_p is better than f(pbest)
      pbest = p;
5) end
6) end
7) gbest = best p in P
8) for each particle p in P do

a) $v_i^{t+1} = \underbrace{v_i^t}_{\text{inertia}} + \underbrace{c_1 u_1^t (pb_i^t - p_i^t)}_{\text{personal influence}} + \underbrace{c_2 u_2^t (gb^t - \dots}_{\text{social influence}}$

10) $p_i^{t+1} = p_i^t + v_i^{t+1}$

11) end
12) end

Output:

iteration 1/20 - Best fitness = 0.081045
iteration 2/20 - Best fitness = 0.081045
iteration 3/20 - Best fitness = 0.006721
:
:
iteration 10/20 - Best fitness = 0.002056
:
iteration 20/20 - Best fitness = 0.000018
Best Solution found:
position: [0.0032145, -0.00268]
fitness: 1.75372e-05

eg: Iteration I

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

value of Cognitive & Social Constant

$c_1 = 2$ & $c_2 = 2$

initial solⁿ are set to 1000

P1 fitness value = $1^2 + 1^2 = 2$

Particle No	initial	Pos	Velocity x	Velocity y	Best	fitness
P1	1	1	0	0	1000	2
P2	-1	1	0	0	1000	2
P3	0.5	-0.5	0	0	100	0.3
P4	1	-1	0	0	100	2
P5	0.5	0.2	0	0	100	0

Iteration-2

pno	initial	Pos	Velocity x	Velocity y	Best	Best Pos x	Best Pos y	final
P1	1	1	-0.75	0.75	2	1	1	2
P2	-1	1	1.25	-0.75	2	-1	1	2
P3	0.5	-0.5	-0.75	0.75	0.5	0.5	-0.5	0.5
P4	1	-1	-0.75	1.25	2	1	-1	2
P5	0.25	0.25	0	0	0.125	0.25	0.25	0.01

Best position: 2.5 Best = 20.250

Code:

```
import numpy as np

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

num_particles = 30
num_dimensions = 5
max_iter = 15
w = 0.7
c1 = 1.5
c2 = 1.5
x_min = -10
x_max = 10

positions = np.random.uniform(x_min, x_max, (num_particles, num_dimensions))
velocities = np.random.uniform(-1, 1, (num_particles, num_dimensions))
personal_best_positions = positions.copy()
personal_best_scores = np.array([objective_function(x) for x in positions])

global_best_index = np.argmin(personal_best_scores)
global_best_position = personal_best_positions[global_best_index].copy()
global_best_score = personal_best_scores[global_best_index]

for iteration in range(max_iter):
    for i in range(num_particles):
        r1 = np.random.rand(num_dimensions)
        r2 = np.random.rand(num_dimensions)
        velocities[i] = (
            w * velocities[i]
            + c1 * r1 * (personal_best_positions[i] - positions[i])
            + c2 * r2 * (global_best_position - positions[i])
        )
        positions[i] = positions[i] + velocities[i]
        positions[i] = np.clip(positions[i], x_min, x_max)
        fitness = objective_function(positions[i])
        if fitness < personal_best_scores[i]:
            personal_best_scores[i] = fitness
            personal_best_positions[i] = positions[i].copy()
    best_particle_index = np.argmin(personal_best_scores)
    if personal_best_scores[best_particle_index] < global_best_score:
        global_best_score = personal_best_scores[best_particle_index]
        global_best_position = personal_best_positions[best_particle_index].copy()

print("Best Solution Found:", global_best_position)
print("Best Fitness Value:", global_best_score)
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

lab-3

Ant Colony optimization for Traveling Salesman Problem

Initialize parameters:
- cities (coordinates)
- No of ants (num_ants)
- pheromone importance (alpha)
- Heuristic importance (beta)
- pheromone evaporation rate (rho)
- no of iteration (iterations)
- initial pheromone on all edges

For each iteration from 1 to iterations
- initialize an empty list to store all tours
- For each ant from 1 to num_ants:
  - start from a random city
  - initialize an empty list to store the tour & visited cities
  - while there are cities to visit:
    - select the next city based on pheromone & heuristic using the probabilities:
      - Calculate probability for each unvisited city:
        P(city) = (pheromone[i][j]^alpha)*(1/distance[i][j]^beta)
      - Normalize probabilities
      - choose next city probabilistically
  - Add the starting city to end of the tour
  - Store tour.
- update pheromones:
  - evaporate pheromones on all edges:
    pheromone[i][j] = pheromone[i][j] * (1 - rho)
  - For each tour in all_tours:
    - Calculate the tour length
    - Deposit pheromone along the edges of tour
      pheromone[i][j] += 1/tour_length

output the best tour & its length after all iterations.

output :-

Best Solution : [0, 4, 3, 1, 0]
Best length : 14

MG
10/10/25

Code:

```
import numpy as np

def ant_colony_optimization(dist_matrix, n_ants, n_iterations, alpha=1, beta=2, rho=0.5, q=100):
    n = len(dist_matrix)
    pheromone = np.ones((n, n))
    visibility = 1 / (dist_matrix + np.eye(n) * 1e10)
    best_length = np.inf
    best_path = None

    for _ in range(n_iterations):
        all_paths = []
        all_lengths = []
        for _ in range(n_ants):
            path = [np.random.randint(n)]
            while len(path) < n:
                i = path[-1]
                prob = (pheromone[i] ** alpha) * (visibility[i] ** beta)
                prob[path] = 0
                prob /= prob.sum()
                next_city = np.random.choice(range(n), p=prob)
                path.append(next_city)
            length = sum(dist_matrix[path[i], path[i+1]] for i in range(n-1)) + dist_matrix[path[-1], path[0]]
            all_paths.append(path)
            all_lengths.append(length)
            if length < best_length:
                best_length = length
                best_path = path
        pheromone *= (1 - rho)
        for path, length in zip(all_paths, all_lengths):
            for i in range(n-1):
                pheromone[path[i], path[i+1]] += q / length
            pheromone[path[-1], path[0]] += q / length
    return best_path, best_length

dist = np.array([
    [0, 2, 9, 10],
    [1, 0, 6, 4],
    [15, 7, 0, 8],
    [6, 3, 12, 0]
])

path, length = ant_colony_optimization(dist, n_ants=10, n_iterations=100)
print("Best path:", path)
print("Best length:", length)
```

Program 4

Cuckoo Search

Lab - 4

Cuckoo Search (CS):

Cuckoo_Search_for_job_scheduling():

1. Initialize n nests with random job orders
2. Define discovery rate pa and MaxIterations
3. For t = 1 to MaxIterations:
    For each nest i:
        Generate new schedule using Levy flights
        Evaluate fitness = TotalCompletionTime(schedule_i)

        if fitness_new < fitness_old:
            replace nest i with new one

    Abandon a fraction pa of worst nests
    Generate new random job order
    Keep the best schedules
4. End for
5) output the job order with the minimum total completion time.

output:

Iteration 1, Best fitness (negative total complet^n time): -152

⋮

Iteration 4, Best fitness (negative total complet^n time): -135

Iteration 5, Best fitness (negative total complet^n time): -134

⋮

Iteration 40, Best fitness (negative total complet^n time): -136

Best job sequence found: [0 6 3 5 4 1 3]

minimum total completion time: 134

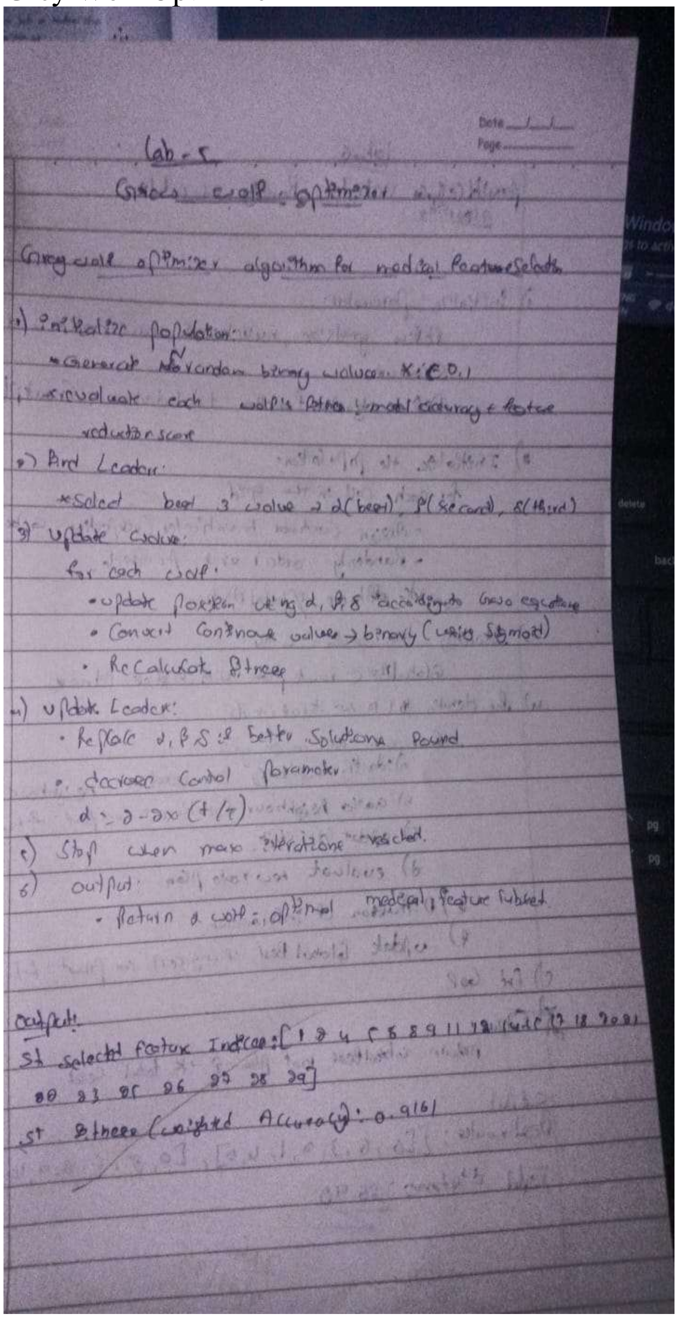
Code:

```
import numpy as np
from scipy.special import gamma  # import gamma from scipy.special

def levy_flight(dim, beta=1.5):
    sigma_u = (gamma(1 + beta) * np.sin(np.pi * beta / 2) /
            gamma((1 + beta) / 2) * beta * np.power(2, (beta - 1) / 2)) ** (1 / beta)
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, 1, dim)
```

```
    step = u / np.power(np.abs(v), 1 / beta)
    return step

def cuckoo_search(objective_function, n, dim, Pa=0.25, MaxGen=100, bounds=(-5, 5)):
    nests = np.random.uniform(bounds[0], bounds[1], (n, dim))
    fitness = np.apply_along_axis(objective_function, 1, nests)
    best_solution = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for gen in range(MaxGen):
        new_nests = np.copy(nests)
        for i in range(n):
            step = levy_flight(dim)
            new_nests[i] = nests[i] + step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])
        new_fitness = np.apply_along_axis(objective_function, 1, new_nests)
        better_nests = new_fitness < fitness
        nests[better_nests] = new_nests[better_nests]
        fitness[better_nests] = new_fitness[better_nests]
        forget_idx = np.random.rand(n) < Pa
        nests[forget_idx] = np.random.uniform(bounds[0], bounds[1], (np.sum(forget_idx), dim))
        fitness = np.apply_along_axis(objective_function, 1, nests)
        current_best_fitness = np.min(fitness)
        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_solution = nests[np.argmin(fitness)]
        print(f"Generation {gen + 1}/{MaxGen}, Best Fitness: {best_fitness}")
    return best_solution, best_fitness

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

n = 30
dim = 5
MaxGen = 100
Pa = 0.25

best_solution, best_fitness = cuckoo_search(sphere_function, n, dim, Pa, MaxGen)

print("\nBest Solution Found: ", best_solution)
print("Best Fitness Value: ", best_fitness)
```

Program 5

Grey Wolf Optimizer



Date __/__/__

Page ______

Lab - 5

Grey wolf optimizer

Grey wolf optimizer algorithm for medical feature selection

1) Initialize population:
   * Generate N random binary wolves $X_i \in \{0,1\}$
   * Evaluate each wolf's fitness (model accuracy + feature reduction score)

2) Find Leader:
   * Select best 3 wolves → α (best), β (second), δ (third)

3) Update wolves:
   for each wolf:
   * update position using α, β, δ according to GWO equations
   * Convert continuous values → binary (using sigmoid)
   * Recalculate fitness

4) Update Leader:
   * Replace α, β, δ if better solutions found
   * decrease control parameter
     $a = 2 - 2 \times (t/T)$

5) Stop when max iterations reached.

6) Output:
   * Return α wolf = optimal medical feature subset

Output:

Selected feature Indices: [1 2 4 5 6 8 9 11 12 14 15 17 18 20 21 22 23 25 26 27 28 29]

Fitness (weighted Accuracy): 0.9161

Code:

```
import numpy as np

def gwo(objective_function, n, dim, max_gen, lb, ub):
    wolves = np.random.uniform(lb, ub, (n, dim))
    fitness = np.apply_along_axis(objective_function, 1, wolves)
    alpha, beta, delta = np.copy(wolves), np.copy(wolves), np.copy(wolves)
    alpha_f, beta_f, delta_f = np.copy(fitness), np.copy(fitness), np.copy(fitness)

    for gen in range(max_gen):
        sorted_idx = np.argsort(fitness)
        alpha, beta, delta = wolves[sorted_idx[0]], wolves[sorted_idx[1]], wolves[sorted_idx[2]]
        alpha_f, beta_f, delta_f = fitness[sorted_idx[0]], fitness[sorted_idx[1]], fitness[sorted_idx[2]]

        A = 2 * np.random.rand(n, dim) - 1
        C = 2 * np.random.rand(n, dim)
        for i in range(n):
            D_alpha = np.abs(C[i] * alpha - wolves[i])
            D_beta = np.abs(C[i] * beta - wolves[i])
            D_delta = np.abs(C[i] * delta - wolves[i])

            wolves[i] = wolves[i] + A[i] * (D_alpha + D_beta + D_delta) / 3
            wolves[i] = np.clip(wolves[i], lb, ub)

        fitness = np.apply_along_axis(objective_function, 1, wolves)
        print(f"Gen {gen+1}/{max_gen}, Best Fitness: {min(fitness)}")

    return alpha, alpha_f

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

n = 30
dim = 5
max_gen = 50
lb, ub = -5, 5

best_solution, best_fitness = gwo(sphere_function, n, dim, max_gen, lb, ub)
print("\nBest Solution Found: ", best_solution)
print("Best Fitness: ", best_fitness)
```

Program 6

Parallel Cellular Algorithm

lab-6

Parallel Cellular Algorithm Vehicle Routing Scheduling

1) Initialize Parameters:

Define gridsize, numvehicle, maxIteration and neighbour structure.

Define objective function: total distance (routeplan).

2) Initialize the population:

for each cell in the grid:

- Assign customers to vehicles (respecting capacity)
- Randomly order visit for vehicle.

Compute fitness = 1/total Distance.

3) identify best solution so far:

GlobalBest ← route plan with highest fitness

4) for iterate = 1 to max iteration do:

for each cell in the grid

a) identify neighbouring cell based on neighbouring structure

b) Gather neighbour solution & compare fitness.

c) update cell state.

d) evaluate new route plan

e) Diffusion of information:

f) update GlobalBest if any cell improved fitness.

5) End loop

6) Output:

return GlobalBest route plan & its total cost.

Output

Best route: [[0, 6, 3, 9, 1, 4, 0], [0, 8, 5, 2, 9, 10, 0]]

Total distance: 86.99

Code: import numpy as np

```
def total_distance(routes, dist_matrix):
    dist = 0
    for route in routes:
        for i in range(len(route)-1):
            dist += dist_matrix[route[i], route[i+1]]
    return dist

def generate_random_routes(num_customers, num_vehicles):
    customers = np.arange(1, num_customers + 1)
    np.random.shuffle(customers)
    splits = np.array_split(customers, num_vehicles)
    return [[0] + list(s) + [0] for s in splits if len(s) > 0]

def crossover(r1, r2):
    flat1 = np.concatenate([x[1:-1] for x in r1 if len(x)>2]) if r1 else []
    flat2 = np.concatenate([x[1:-1] for x in r2 if len(x)>2]) if r2 else []
    if len(flat1)==0 or len(flat2)==0:
        return r1 if len(flat1)>0 else r2
    point = np.random.randint(1, len(flat1))
    child_seq = list(flat1[:point]) + [x for x in flat2 if x not in flat1[:point]]
    splits = np.array_split(child_seq, 2)
    return [[0]+list(s)+[0] for s in splits if len(s)>0]

def mutate(routes, rate=0.2):
    for route in routes:
        if np.random.rand() < rate and len(route) > 3:
            i, j = np.random.choice(range(1, len(route)-1), 2, replace=False)
            route[i], route[j] = route[j], route[i]
    return routes

def local_optimize(routes, dist_matrix):
    for route in routes:
        improved = True
        while improved:
            improved = False
            for i in range(1, len(route)-2):
                for j in range(i+1, len(route)-1):
                    a, b, c, d = route[i-1], route[i], route[j], route[j+1]
                    before = dist_matrix[a,b]+dist_matrix[c,d]
                    after = dist_matrix[a,c]+dist_matrix[b,d]
                    if after < before:
                        route[i:j+1] = route[i:j+1][::-1]
                        improved = True
    return routes

def create_distance_matrix(num_customers):
    coords = np.random.rand(num_customers + 1, 2) * 20
```

```
    return np.sqrt(((coords[:, None, :] - coords[None, :, :]) ** 2).sum(-1))

def parallel_cellular_vrp(num_customers=10, num_vehicles=2, grid_size=(3,3), iterations=50):
    dist_matrix = create_distance_matrix(num_customers)
    grid = np.empty(grid_size, dtype=object)
    fit = np.zeros(grid_size)

    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            routes = generate_random_routes(num_customers, num_vehicles)
            grid[i,j] = routes
            fit[i,j] = 1 / (1 + total_distance(routes, dist_matrix))

    best = grid[np.unravel_index(np.argmax(fit), fit.shape)]

    for _ in range(iterations):
        new_grid = np.empty_like(grid)
        for i in range(grid_size[0]):
            for j in range(grid_size[1]):
                neighbors = []
                for di in [-1,0,1]:
                    for dj in [-1,0,1]:
                        if di==0 and dj==0: continue
                        ni, nj = (i+di)%grid_size[0], (j+dj)%grid_size[1]
                        neighbors.append(grid[ni,nj])
                best_neighbor = max(neighbors, key=lambda r: 1/(1+total_distance(r, dist_matrix)))
                child = crossover(grid[i,j], best_neighbor)
                child = mutate(child)
                child = local_optimize(child, dist_matrix)
                new_fit = 1 / (1 + total_distance(child, dist_matrix))
                if new_fit > fit[i,j]:
                    new_grid[i,j] = child
                    fit[i,j] = new_fit
                else:
                    new_grid[i,j] = grid[i,j]
        grid = new_grid
        current_best = grid[np.unravel_index(np.argmax(fit), fit.shape)]
        if (1/(1+total_distance(current_best, dist_matrix))) > (1/(1+total_distance(best, dist_matrix))):
            best = current_best

    return best, total_distance(best, dist_matrix)

best_routes, best_cost = parallel_cellular_vrp()
print("Best Routes:", best_routes)
print("Total Distance:", round(best_cost, 2))
```

Program 7

Optimization via Gene Expression Algorithm

Lab-7

Optimization via Gene expression algo

Total 6 phase

- → initialization
- → fitness assignment
- → Selection
- → Cross over
- → mutation
- → Gene expression
- → Termination

Step 1:- fitness $(x) = x^2$

1) select encoding technique 0 to 31, use chromosome of fixed length with (variable, constraints, function)

2) Initial population:-

	Initial Chromosome	Phenotype	value	fitness	probability
1	+x x	x	12	144	0.1247
2	+x x	2x	25	625	0.541
3	-x	x	5	25	0.0216
4	-x2	x-2	19	361	0.3125

$\Sigma f(x) = 1155$

Avg = 288.75

actual	expected
[illegible]	0.5
2	2.1
0	0.08
1	1.25

3) Selection of mating pool

	Selected Chromosome	Crossover pool	offspring	phenotype	value
1	+x x	2	+x x	x(x+1)	13
2	+x x	1	+x x	2x	24
3	+x x	3	+x -	x(x-)	27
4	-x2	1	+x2	x+2	17

fitness

169

576

729

289

4) Crossover:- perform crossover randomly choose gene position max fitness after crossover = 729.

5) Mutation:-

	offspring after mutation	mutation applied	offspring after mutation	Phenotype	x value
1	+x x	+x -	+x -	x^2(x-)	29
2	+x x	None	+x x	2x	24
3	+x -	-x	+x x	x-	27
4	+x2	none	+x2	x+2	20

fitness

841

576

729

400

6) Gene expression & evolution

Decode each genotype → phenotype

Calculate fitness

$\Sigma f(x) = 2546$

Avg = 636.5

max = 841

7) Iterate until convergence repeat steps 3-6 until fitness improvement is negligible/generation limit has reached.

Pseudo code:-

- Define fitness evolution func,
- create population
- Define parameters
- Select mating pool
- mutation after mating
- Gene expression and evolution
- iterate
- output Best value

output:-

Genes = [29.53, 29.82, 29.34, 28.54, 15.81, 21.08, 23.13, 30.81, 22.51, 26.22]

x = 26.37

f(x) = 695.45.

Code:

```
import random
import math

def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 2

POPULATION_SIZE = 6
GENE_LENGTH = 10
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
GENERATIONS = 20
DOMAIN = (-1, 2)

def random_gene():
    return random.uniform(DOMAIN[0], DOMAIN[1])

def create_chromosome():
    return [random_gene() for _ in range(GENE_LENGTH)]

def initialize_population(size):
    return [create_chromosome() for _ in range(size)]

def evaluate_population(population):
    return [fitness_function(express_gene(chrom)) for chrom in population]

def express_gene(chromosome):
    return sum(chromosome) / len(chromosome)

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual
    return random.choice(population)

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

def mutate(chromosome):
    new_chromosome = []
```

```
    for gene in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome.append(random_gene())
        else:
            new_chromosome.append(gene)
    return new_chromosome

def gene_expression_algorithm():
    population = initialize_population(POPULATION_SIZE)
    best_solution = None
    best_fitness = float("-inf")

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        for i, chrom in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = chrom[:]

        print(f"Generation {generation+1}: Best Fitness = {best_fitness:.4f}, Best x =
{express_gene(best_solution):.4f}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

    print("\nBest solution found:")
    print(f"Genes: {best_solution}")
    x_value = express_gene(best_solution)
    print(f"x = {x_value:.4f}")
    print(f"f(x) = {fitness_function(x_value):.4f}")

if __name__ == "__main__":
    gene_expression_algorithm()
```