

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sareddy Poojya Sree (1BM23CS303)

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 **Sareddy Poojya Sree (1BM23CS303)**, 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

Sl. No.	Date	Experiment Title	Page No.
1	29/8/25	Genetic Algorithm	4
2	12/9/25	Particle Swarm Optimization	7
3	10/10/25	Ant Colony Optimization	9
4	17/10/25	Cuckoo Search Optimization	12
5	17/10/25	Grey Wolf Optimization	15
6	7/11/25	Parallel Cellular Algorithm	19
7	29/8/25	Gene Expression Algorithm	23

Github Link:

[poojya100/1BM23CS303_BIS_LAB](https://github.com/poojya100/1BM23CS303_BIS_LAB)

Program 1

Problem statement: A salesman must visit a given list of n -cities exactly once and return to the starting city. The distance between each pair of cities is known. The goal is to determine the shortest possible route that visits all cities.

Use Genetic Algorithm to find a near-optimal solution to the Travelling Salesman Problem by evolving candidate routes toward the minimum total travel distance.

Algorithm:

Genetic Algorithm:

$$f(x) = x^2$$

Steps:

1. selecting encoding techniques $\rightarrow 0$ to 31 .
2. select initial population: $- "4"$

String no.	Initial population	x value	fitness $f(x) = x^2$	Prob. $f(x)/\sum f(x)$	% Prob	Expected count	Actual count
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	0.086	0
4	10011	19	361	0.3125	31.25	1.25	1

Sum 1155

Avg 288.75

Max 625

3. select Mating pool:

String no.	Mating pool	crossover point	Off spring 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

Sum 1155

Avg 288.75

Max 625

4) Crossover: Random 4 & 2
 Max value - 729

5) Mutation :

String no	offspring after crossover	Mutation chromosome for flippings	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
Sum					2546
Avg					630.5
Max.					841.

Pseudocode:

Output:

Generation 0: Best fitness = 729, Best individual = 27

Generation 1: Best fitness = 729, Best individual = 27

Generation 2: Best fitness = 729, Best individual = 27

Generation 3: Best fitness = 729, Best individual = 27

Generation 18: Best fitness = 729, Best individual = 27

Generation 19: Best fitness = 729, Best individual = 27

Best solution: $x = 27$, $f(x) = 729$

Sum
29/8/25

Code:

```
import random
# Problem parameters
CHROMOSOME_LENGTH = 5 # 5 bits to represent numbers 0-31
POPULATION_SIZE = 10
GENERATIONS = 20
CROSSOVER_RATE = 0.7
MUTATION_RATE = 0.01
# Decode binary chromosome to integer
def decode(chromosome):
    return int("".join(str(bit) for bit in chromosome), 2)
# Fitness function: maximize x^2
def fitness(chromosome):
    x = decode(chromosome)
    return x ** 2
# Generate initial population
def init_population():
    population = []
    for _ in range(POPULATION_SIZE):
        chromosome = [random.randint(0,1) for _ in range(CHROMOSOME_LENGTH)]
        population.append(chromosome)
    return population

# Roulette Wheel Selection
def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for i, fit in enumerate(fitnesses):
        current += fit
        if current > pick:
            return population[i]

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

# Mutation: bit flip
def mutate(chromosome):
    for i in range(CHROMOSOME_LENGTH):
        if random.random() < MUTATION_RATE:
            chromosome[i] = 1 - chromosome[i]
    return chromosome

# Main GA loop
def genetic_algorithm():
    population = init_population()

    for generation in range(GENERATIONS):
        fitnesses = [fitness(chromo) for chromo in population]

        print(f"Generation {generation}: Best fitness = {max(fitnesses)}, Best individual = {decode(population[fitnesses.index(max(fitnesses))])}")
```

```

new_population = []

while len(new_population) < POPULATION_SIZE:
    parent1 = select(population, fitnesses)
    parent2 = select(population, fitnesses)
    child1, child2 = crossover(parent1, parent2)
    child1 = mutate(child1)
    child2 = mutate(child2)
    new_population.extend([child1, child2])

population = new_population[:POPULATION_SIZE]

# Final result
fitnesses = [fitness(chromo) for chromo in population]
best_index = fitnesses.index(max(fitnesses))
best_chromosome = population[best_index]
best_value = decode(best_chromosome)
print(f"Best solution: x = {best_value}, f(x) = {fitness(best_chromosome)}")

if __name__ == "__main__":
    genetic_algorithm()

```

Program 2

Problem Statement:

Training a neural network involves finding an optimal set of weights and biases that minimize prediction error. Traditional gradient-based optimization methods.

Use Particle Swarm Optimization to optimize the weights and biases of a neural network by treating each particle as a potential weight vector and iteratively updating their positions to minimize the network's loss function.

Algorithm:

Lab-2 Particle Swarm Optimisation.

Pseudocode:

1. $P = \text{particle initialization}()$

2. For $i=1$ to max
for each particle p in P do

$$fp = f(p)$$

if fp is better than $f(p_{\text{best}})$
 $p_{\text{best}} = p$

end

end

$g_{\text{best}} = \text{best } p \text{ in } P.$

for each particle p in P do

$$V_i^{t+1} = \underbrace{V_i^t}_{\text{inertia}} + \underbrace{C_1 V_1^t (p_{\text{best}}^t - P_i^t)}_{\text{personal influence}} + \underbrace{C_2 V_2^t (g_{\text{best}}^t - P_i^t)}_{\text{social influence}}$$

$$P_i^{t+1} = P_i^t + V_i^{t+1}$$

end

end.

Output:

Iteration 1/50 | Best value: 0.786857 at $[-0.44260247, -0.7687538668138685]$

Iteration 50/50 | Best value: 0.000000 at $[9.119794577206748e^{-09}, -2.0757413670574333e^{-08}]$

Optimal solution found:

Best position: $[9.119794577206948e^{-9}, -2.0757413670574333e^{-08}]$

Minimal value: $5.140408754217994e^{-16}$

Code:

```
import random
```

```
# Objective (fitness) function: De Jong function
```

```
def fitness_function(position):
```

```
    x, y = position
```

```
    return x**2 + y**2 # minimize this function
```

```
# PSO parameters
```

```
num_particles = 10
```

```
num_iterations = 50
```

```
W = 0.3 # inertia weight (from PDF)
```

```
C1 = 2 # cognitive coefficient
```

```
C2 = 2 # social coefficient
```

```
# Initialize particles and velocities
```

```
particles = [[random.uniform(-10, 10), random.uniform(-10, 10)] for _ in  
range(num_particles)]
```

```
velocities = [[0.0, 0.0] for _ in range(num_particles)]
```

```
# Initialize personal bests
```

```
pbest_positions = [p[:] for p in particles]
```

```
pbest_values = [fitness_function(p) for p in particles]
```

```
# Initialize global best
```

```
gbest_index = pbest_values.index(min(pbest_values))
```

```
gbest_position = pbest_positions[gbest_index][:]
```

```
gbest_value = pbest_values[gbest_index]
```

```
# PSO main loop
```

```
for iteration in range(num_iterations):
```

```
    for i in range(num_particles):
```

```
        r1, r2 = random.random(), random.random()
```

```
        # Update velocity
```

```
        velocities[i][0] = (W * velocities[i][0] +  
                             C1 * r1 * (pbest_positions[i][0] - particles[i][0]) +  
                             C2 * r2 * (gbest_position[0] - particles[i][0]))
```

```
        velocities[i][1] = (W * velocities[i][1] +  
                             C1 * r1 * (pbest_positions[i][1] - particles[i][1]) +  
                             C2 * r2 * (gbest_position[1] - particles[i][1]))
```

```

# Update position
particles[i][0] += velocities[i][0]
particles[i][1] += velocities[i][1]

# Evaluate fitness
current_value = fitness_function(particles[i])

# Update personal best
if current_value < pbest_values[i]:
    pbest_positions[i] = particles[i][:]
    pbest_values[i] = current_value

# Update global best
if current_value < gbest_value:
    gbest_value = current_value
    gbest_position = particles[i][:]

print(f"Iteration {iteration+1}/{num_iterations} | Best Value: {gbest_value:.6f} at {gbest_position}")

print("\nOptimal Solution Found:")
print(f"Best Position: {gbest_position}")
print(f"Minimum Value: {gbest_value}")

```

Program 3

Problem Statement:

- In a communication network, data packets must be routed from a source node to a destination node through multiple possible paths. As the network grows larger and more dynamic, finding the shortest and least congested path becomes increasingly complex for traditional deterministic routing algorithms.

Use Ant Colony Optimization to compute the optimal or near-optimal routing path between nodes in a network.

Algorithm:

Ant Colony Optimization:

Initialize pheromone $\tau(i, j) = \tau_0$ for all edges set parameters α, β, p, Q

BestTourlength = $+\infty$

Repeat for each iteration:

for each ant:

place ant at random city

while tour not complete:

choose next city j with prob.:

$$P[i, j] = [\tau(i, j)]^\alpha * [\eta(i, j)]^\beta$$

move to city j

complete tour length LK

update best tour if $LK < \text{BestTourlength}$

Evaporate pheromone: $\tau(i, j) = (1 - p) * \tau(i, j)$

For each ant:

For each edge (i, j) in its tour:

$$\tau(i, j) + Q / LK$$

until stopping condition met

Return BestTour, BestTourlength

Output:

Iteration 1: Best iteration = 23.0

Iteration 2: Best iteration = 21.0

Iteration 3: Best iteration = 20.0

Iteration 4: Best iteration = 20.0

Iteration 5: Best iteration = 20.0

Code:

```
import numpy as np

class AntColony:
    def __init__(self, distances, n_ants, n_best, n_iterations, decay, alpha=1,
beta=1):
        """
        distances: 2D numpy array of distances between cities
        n_ants: number of ants per iteration
        n_best: number of best ants who deposit pheromone
        n_iterations: number of iterations
        decay: rate of pheromone evaporation (0 < decay < 1)
        alpha: pheromone importance
        beta: distance importance (heuristic)
        """
        self.distances = distances
        self.pheromone = np.ones(self.distances.shape) / len(distances)
        self.all_inds = range(len(distances))
        self.n_ants = n_ants
        self.n_best = n_best
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

    def run(self):
        shortest_path = None
        all_time_shortest_path = ("placeholder", np.inf)
        for i in range(self.n_iterations):
            all_paths = self.gen_all_paths()
            self.spread_pheromone(all_paths, self.n_best,
shortest_path=shortest_path)
            shortest_path = min(all_paths, key=lambda x: x[1])
            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path
            self.pheromone *= (1 - self.decay)
        return all_time_shortest_path

    def spread_pheromone(self, all_paths, n_best, shortest_path):
        sorted_paths = sorted(all_paths, key=lambda x: x[1])
        for path, dist in sorted_paths[:n_best]:
            for move in path:
                self.pheromone[move] += 1.0 / self.distances[move]

    def gen_path_dist(self, path):
        total_dist = 0
        for i in range(len(path)):
            from_city = path[i][0]
            to_city = path[i][1]
            total_dist += self.distances[from_city][to_city]
        return total_dist

    def gen_all_paths(self):
        all_paths = []
        for i in range(self.n_ants):
            path = self.gen_path(0) # start from city 0
            dist = self.gen_path_dist(path)
            all_paths.append((path, dist))
        return all_paths
```

```

def gen_path(self, start):
    path = []
    visited = set()
    visited.add(start)
    prev = start
    for _ in range(len(self.distances) - 1):
        move = self.pick_move(self.pheromone[prev], self.distances[prev],
visited)
        path.append((prev, move))
        prev = move
        visited.add(move)
    path.append((prev, start)) # return to start
    return path

def pick_move(self, pheromone, dist, visited):
    pheromone = np.copy(pheromone)
    pheromone[list(visited)] = 0

    row = pheromone ** self.alpha * (( 1.0 / dist) ** self.beta)
    norm_row = row / row.sum()
    move = np.random.choice(self.all_inds, 1, p=norm_row)[0]
    return move

# Example usage:
if __name__ == "__main__":
    # Distance matrix (symmetric) between 4 cities
    distances = np.array([
        [np.inf, 2, 2, 5],
        [2, np.inf, 3, 4],
        [2, 3, np.inf, 1],
        [5, 4, 1, np.inf]
    ])

    ant_colony = AntColony(distances, n_ants=10, n_best=3, n_iterations=100,
decay=0.1, alpha=1, beta=2)
    shortest_path = ant_colony.run()
    print("Shortest path:", shortest_path[0])
    print("Distance:", shortest_path[1])

```

Program 4

Problem Statement:

Many engineering design problems, such as designing a spring, a gear system, or a pressure vessel, require determining a set of parameters that minimize cost while satisfying mechanical, safety, and performance constraints.

Use Cuckoo Search Optimization to determine the optimal design parameters for an engineering system.

Algorithm:

Cuckoo search optimization:

Application: Wind turbine Blade design Optimization
[to find optimal blade pitch angle to get max. power from wind turbine]

Step 1: Create a function to calculate power coefficient C_p based on blade pitch angle

Step 2: Initialise randomly, generate multiple candidate solutions within pitch angle bounds.

Step 3: Evaluate fitness by calculating C_p for each and finding best one.

Step 4: Iterate:

- * For each nest, guarantee new solution using Levy flights around the best nest.
- * Replace old nest if new ones has better fitness.
- * Within a small probability, abandon some nests and create a new random ones.
- * Updates best solution found.

Step 5: Return result, output pitch angle corresponding to highest C_p .

O/p:

Optimised blade pitch angle : 9.998 deg.
maximum power coefficient (C_p) : 100.0000

Code:

```
import numpy as np
import math

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

# Generate initial population (nests)
def initialize_nests(num_nests, dim, lower_bound, upper_bound):
    return np.random.uniform(lower_bound, upper_bound, size=(num_nests, dim))

# Levy flight for generating new solutions
def levy_flight(Lambda, size):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
              (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2))) **
    (1 / Lambda)
    u = np.random.randn(*size) * sigma
    v = np.random.randn(*size)
    step = u / np.abs(v) ** (1 / Lambda)
    return step

# Cuckoo Search Algorithm
def cuckoo_search(num_nests=25, dim=2, lower_bound=-10, upper_bound=10,
                  pa=0.25, max_iter=100):

    nests = initialize_nests(num_nests, dim, lower_bound, upper_bound)
    fitness = np.apply_along_axis(objective_function, 1, nests)

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

    for t in range(max_iter):
        # Generate new solutions via Levy flights
        new_nests = nests + 0.01 * levy_flight(1.5, nests.shape) * (nests -
best_nest)
        new_nests = np.clip(new_nests, lower_bound, upper_bound)

        # Evaluate new solutions
        new_fitness = np.apply_along_axis(objective_function, 1, new_nests)

        # Replace nests if better
        mask = new_fitness < fitness
        nests[mask] = new_nests[mask]
        fitness[mask] = new_fitness[mask]

        # Discovery and randomization
        rand = np.random.rand(num_nests, dim)
        new_nests = np.where(rand > pa, nests,
                              initialize_nests(num_nests, dim, lower_bound,
upper_bound))

        new_fitness = np.apply_along_axis(objective_function, 1, new_nests)
        mask = new_fitness < fitness
        nests[mask] = new_nests[mask]
        fitness[mask] = new_fitness[mask]

        # Update best solution
        if np.min(fitness) < best_fitness:
```

```

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

    print(f"Iteration {t+1}/{max_iter} | Best Fitness: {best_fitness:.6f}")

    return best_nest, best_fitness

# Run Cuckoo Search
best_solution, best_value = cuckoo_search()
print("\nBest solution found:", best_solution)
print("Best fitness value:", best_value)

```

Program 5

Problem Statement:

Support Vector Machines (SVMs) require optimal selection of hyperparameters—such as the regularization parameter C , kernel parameter γ , and kernel type—to achieve high classification accuracy.

Use Grey Wolf Optimization to automatically determine the optimal SVM hyperparameters by modelling each wolf as a candidate solution in the (C, γ) search space. The wolves will follow the leadership hierarchy (alpha, beta, delta) and encircling–hunting behavior to explore and exploit the parameter space.

Algorithm:

Grey Wolf Optimiser:

Application: SVM hyperparameter tuning.

Algorithm:

Step 1: Prepare data — Scale features and split into training and testing datasets.

Step 2: Define fitness function — $\text{fitness}(\text{params}) = \text{accuracy}(\text{sum with params})$ on test / validation set (minimize this).

Step 3: Initialise wolves: randomly generate N candidate solutions (each = $[c, \gamma]$) within bounds.

Step 4: Evaluate Fitness function: Compute fitness for every wolf, identify α (best), β (2nd), γ (3rd).

Step 5: loop (for $t = 1 \dots T$)

- Update exploration coefficient $a = 2 - 2t/T$
- For each wolf and each parameters dimension, compute A and C using random r_1, r_2 from x_1, x_2, x_3 from α, β, γ ; set new position = $(x_1 + A x_2 + C x_3) / 3$
- Clip positions to bound.
- Re-evaluate fitness and update α, β, γ

Step 6: Stop: — after T iterations, take α 's position as best (c^*, γ^*) .

Step 7: Train final SVM - Train SVM on training set using (c^*, γ^*) and evaluate on test set.

Step 8: Return results: Report best parameters and final accuracy (or other metrics).

O/P: Iteration 25/25 | Best fitness : 0.035

Best Parameters found:

$C = 19.4821$, $\gamma = 0.0123$

final accuracy : 0.9650.

Code:

```
import numpy as np

# Objective function to minimize (you can modify this)
def objective_function(x):
    # Example: Sphere function ->  $f(x) = \sum(x^2)$ 
    return np.sum(x**2)

# Grey Wolf Optimizer function
def grey_wolf_optimizer(num_wolves=30, dim=2, max_iter=50, lower_bound=-10, upper_bound=10):
    # Initialize the positions of wolves
    wolves = np.random.uniform(lower_bound, upper_bound, (num_wolves, dim))

    # Initialize alpha, beta, delta wolves
    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)

    Alpha_score = float("inf")
    Beta_score = float("inf")
    Delta_score = float("inf")

    # Main optimization loop
    for t in range(max_iter):
        for i in range(num_wolves):
            # Keep wolves inside the search space
            wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)

            # Calculate fitness
            fitness = objective_function(wolves[i])
```

```

        # Update Alpha, Beta, Delta
        if fitness < Alpha_score:
            Delta_score = Beta_score
            Delta_pos = Beta_pos.copy()
            Beta_score = Alpha_score
            Beta_pos = Alpha_pos.copy()
            Alpha_score = fitness
            Alpha_pos = wolves[i].copy()
        elif fitness < Beta_score:
            Delta_score = Beta_score
            Delta_pos = Beta_pos.copy()
            Beta_score = fitness
            Beta_pos = wolves[i].copy()
        elif fitness < Delta_score:
            Delta_score = fitness
            Delta_pos = wolves[i].copy()

    # Linearly decreasing parameter a from 2 to 0
    a = 2 - t * (2 / max_iter)

    # Update the position of each wolf
    for i in range(num_wolves):
        for j in range(dim):
            r1 = np.random.rand()
            r2 = np.random.rand()

            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * Alpha_pos[j] - wolves[i][j])
            X1 = Alpha_pos[j] - A1 * D_alpha

            r1 = np.random.rand()
            r2 = np.random.rand()
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * Beta_pos[j] - wolves[i][j])
            X2 = Beta_pos[j] - A2 * D_beta

            r1 = np.random.rand()
            r2 = np.random.rand()
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * Delta_pos[j] - wolves[i][j])
            X3 = Delta_pos[j] - A3 * D_delta

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

    print(f"Iteration {t+1}/{max_iter} | Best Fitness: {Alpha_score:.6f}")

    return Alpha_pos, Alpha_score

# Run the optimizer
best_position, best_score = grey_wolf_optimizer()
print("\nBest solution found:", best_position)
print("Best fitness value:", best_score)

```

Program 6

Problem Statement:

Modern communication networks require routing algorithms that can adapt quickly to changes in traffic load, link failures, and congestion. Traditional centralized routing strategies may suffer from slow updates, high computational cost, and poor scalability as network size increases.

Use a Parallel Cellular Algorithm to compute optimal routing paths in a dynamic communication network. Each cell in the cellular grid represents a router or network node and updates its routing information based on local interactions with neighbouring cells.

Algorithm:

Parallel Cellular Algorithm.

Application: Network Routing.

Algorithm:

1. Define problem: Represent network as a grid of cells.
2. Initialize parameters: Define neighbourhood, cost metrics.
3. Initialize population: Assign initial path costs.
4. Evaluate fitness: Compute routing cost per node.
5. Update states: Update using neighbours minimum cost.
6. Iterate: Repeat until convergence.
7. Output result: Extract shortest path.

Output:

Shortest Path from Source to Destination:
 $(0,0) \rightarrow (1,0) \rightarrow (2,0) \rightarrow (2,1) \rightarrow (3,1) \rightarrow$
 $(3,2) \rightarrow (4,2) \rightarrow (4,3) \rightarrow (4,4)$

Path cost: 33.0.

Code:

```
import numpy as np

# -----
# Step 1: Define Network as Grid
# -----
GRID_SIZE = 5 # 5x5 network grid
MAX_ITER = 100
INF = 1e9

# Define source and destination
source = (0, 0)
destination = (4, 4)

# Create cost matrix for each link (random or distance-based)
np.random.seed(42)
cost_matrix = np.random.randint(1, 10, size=(GRID_SIZE, GRID_SIZE))

# Initialize state matrix (each cell's current best cost to destination)
state = np.full((GRID_SIZE, GRID_SIZE), INF)
state[destination] = 0 # destination cost = 0

# Define 4-neighborhood (up, down, left, right)
neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# -----
# Step 2: Define Helper Function
# -----
def get_neighbors(i, j):
    """Return valid neighboring cells"""
    valid_neighbors = []
    for dx, dy in neighbors:
        ni, nj = i + dx, j + dy
        if 0 <= ni < GRID_SIZE and 0 <= nj < GRID_SIZE:
            valid_neighbors.append((ni, nj))
    return valid_neighbors

# -----
# Step 3: PCA Iteration for Routing
# -----
for iteration in range(MAX_ITER):
    new_state = state.copy()
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            if (i, j) == destination:
                continue
            neighbor_costs = []
```

```

        for ni, nj in get_neighbors(i, j):
            # Update rule: min(cost to neighbor + neighbor's state)
            total_cost = cost_matrix[ni, nj] + state[ni, nj]
            neighbor_costs.append(total_cost)
        if neighbor_costs:
            new_state[i, j] = min(neighbor_costs)
    # Check for convergence
    if np.allclose(new_state, state):
        print(f"Converged after {iteration} iterations.")
        break
    state = new_state

# -----
# Step 4: Extract Path from Source
# -----
path = [source]
current = source
while current != destination:
    i, j = current
    nbs = get_neighbors(i, j)
    next_cell = min(nbs, key=lambda n: state[n])
    path.append(next_cell)
    current = next_cell

# -----
# Step 5: Display Results
# -----
print("Final Routing Cost Grid:")
print(np.round(state, 2))
print("\nShortest Path from Source to Destination:")
print(" → ".join([str(p) for p in path]))
print(f"\nTotal Path Cost: {state[source]}")

```

Program 7

Problem Statement:

Machine learning models often perform poorly when the original input features do not sufficiently capture the underlying patterns in the data. Manually engineering new features is time-consuming and requires domain expertise.

Use the Gene Expression and Evaluation Algorithm to automatically construct new features from existing input variables for a supervised learning task.

Algorithm:

Lab - 7

GEA.

Gene expression Algorithm: 6 Main Phases.

- Initialisation
- Fitness Assignment.
- selection
- Cross over
- Gene expression
- Termination

Steps: Fitness (x) = x^2

1) select encoding Technique: 0 - 31

Use chromosome of fixed length with terminals (variables, constants and functions (+, -, *, /)).

2) Initial Population:

S.No	Initial chromosome	Phenotype	Value	Fitness	P
1.	x^2	x^2	12	144	0.1247
2.	$2x$	$2x$	25	625	0.541
3.	x	x	5	25	0.0216
4.	$x-2$	$x-2$	19	361	0.8125

$$\sum P(x) = 1155$$

$$Avg = 288.75$$

Actual cost	Exp. cost
1	0.5
2	2.1
0	0.08
1	1.25

3) selection of Mating Pool.

S.No	selected chromosome	crossover point	offspring	Phenotype
1	*xx	2	*x+	$x^*(x+...)$
2	+xx	1	+xx	2x
3	+x-	3	+x-	$x+(x+...)$
4	-x2	1	+x2	$x+2$

x values	Fitness
13	169
24	576
27	729
17	289

4) Crossover:

Perform crossover randomly chosen gene positions (not raw bits)

Max fitness after crossover = 729.

5) Mutation:

S.No	offspring before mutation	Mutation Applied	Offspring after mutation	Phenotype
1	*x+	- → +	+x-	$x^*(x+...)$
2	+xx	None	+xx	2x
3	+x-	- → *	+x*	$x+(x+...)$
4	+x2	None	+x2	$x+2$

x value	Fitness f(x)
29	841
24	576
27	729
20	400

Code:

```
import random
# Define the function to maximize
def fitness_function(x):
    return x ** 2
# Convert binary string to integer
def decode(chromosome):
    return int(chromosome, 2)
# Evaluate fitness for the entire population
def evaluate_population(population):
    return [fitness_function(decode(individual)) for individual in population]
# Selection: Roulette Wheel
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

# Crossover: 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

# Mutation: Flip random bit
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

# Get user input for initial population
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

# Run the Genetic Algorithm
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)
        # Update best solution
        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))}")

# Parameters
POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5
MUTATION_RATE = 0.01

```

```
CROSSOVER_RATE = 0.8  
GENERATIONS = 20
```

```
# Run it  
if __name__ == "__main__":  
    genetic_algorithm()
```