VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Bramha Anilkumar Bajannavar (1BM23CS071)

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 Dec-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 **Bramha Anilkumar Bajannavar (1BM23CS071),** 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.

Raghavendra C K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/BramhaBajannavar/Bio_Inspired_systems

Program 1

Genetic Algorithm for Optimization Problems:

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

Algorithm:

Page No. | Teacher Sign / Remarks

Date 18/8/25
Page

1. Genetic Algorithm

1) Selecting initial population.
2) Calculate the fitness.
3) Selecting mating pool
4) Crossover
5) Mutation.

Eg:-

Step 1:- Select randomly $x \rightarrow 0-31$
let 12, 25, 5, 19.

Step 2:-

String no	Initial pop^{n}	x value	Fitness func^{n} $f(x^2)$	prob	% prob	Expected Output	Actual value.
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	(625)	(0.5411)	54.11	2.16	2
3	00101	5	25	0.0260	2.16	0.08	0
4	10011	19	361	0.3126	31.26	1.25	1
			$\sum f(x^2)$				
Sum			1155	1.0	100	4	
Avg			288.75	0.25	25	1	
Max			625	0.5411	54.11	2.16	

$$\text{prob} = \frac{f(x)}{\sum f(x)}$$

$$\text{Expected output} = \frac{f(x)}{\text{Avg}\left(\sum f(x)\right)}$$

Actual output

Round Exp output

Date / /
Page

8) Selecting mating pool.

Mating pool	Crossover point	offspring	x value	fitness $f(x)=x^2$
01100	4	01101	13	169
11001		11000	24	576
11001	2	11011	27	729
10011		10001	17	289
Sum				1763
Avg				440.75
Max				729.

4) Crossover point is chosen randomly

Mutation.

Offspring	Mutation Chromosome	Offspring mutation	x value	fitness
01101	10000	11101	29	841
11000	00000	11000	24	576
11011	00000	11011	27	729
10001	00101	10100	20	400
Sum				2546
Avg				636.5
Max.				841

Date __/__/__
Page _____

TSP using GA.

fitness
fun.1 (fx)
169
576
729
289
1763
440.75
729.

fitness
841
576
729
400
2546
636.5
841

Pseudo code -

1. Initialise population
   - Generate N random routes, each route is a permutation of cities.

2. Evaluate fitness
   - For each route in popula
   - calculate total dist and fitness.
     fitness = 1/distance

3. Repeat until stopping criteria
   a. Selection
      - Select a subset of individuals (parents) based on fitness (tournament selection)
      - More fit routes have higher chance of getting selected

   b. Crossover
      - For each pair of parents
        - Combine them and produce offspring
        - Ensure offspring are valid

   c. Mutation.
      - For each offspring,
        - with a small mutation swap two cities.

   d. Evaluate fitness of offspring.

4. Return the best solution.

Output:

Input adjacency matrix:

[0, 2, 9, 10],
[1, 0, 6, 4]
[15 7 0 8]
[6 3 12 0]

Generation 1: Best dist = 21.00
Generation 2: Best dist = 21.00
:

Generation 15: Best dist = 15.00
Best route found [2, 3, 1, 0, 0]
Distance of best route: 21

Pseudo code:-

fitness function:-
return 1/distance_route

def distance_route(x1, x2):
return Math.dist(x1, x2)

Date __/__/__
Page _____

```
def selection(population):
    tournament_size = 3
    selected = []
    for _ in range(population):
        tournament = random.sample(tournament_size)

    return selected


def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(num), 2))
    p2_index = 0
    if child[i] = None:
        while parent2[p2_index] in child:
            p2_index += 1

    return child


def mutate(route):
    if random.random() < mutation_rate:
        route[i], route[j] = route[j], route[i]
```

Sep Rev 2nd

Code:

```
import random

def create_chromosome(length):
    return [random.choice([0, 1]) for _ in range(length)]

def calculate_fitness(chromosome):
    """Calculates the fitness (number of '1's) of a chromosome."""
    return sum(chromosome)

def select_parents(population, num_parents):
    """Selects parents based on their fitness (roulette wheel selection)."""
    total_fitness = sum(calculate_fitness(c) for c in population)
    selection_probs = [calculate_fitness(c) / total_fitness for c in population]

    parents = random.choices(population, weights=selection_probs, k=num_parents)
    return parents

def crossover(parent1, parent2):
    """Performs single-point crossover."""
    crossover_point = random.randint(1, len(parent1) - 1)

    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]

    return child1, child2

def mutate(chromosome, mutation_rate):
    """Performs mutation by flipping bits with a given probability."""
    for i in range(len(chromosome)):
        if random.random() < mutation_rate:
            chromosome[i] = 1 - chromosome[i]  # Flip the bit
    return chromosome

def genetic_algorithm(chromosome_length, population_size, generations, mutation_rate):
    """Main genetic algorithm loop."""
    # 1. Initialization
    population = [create_chromosome(chromosome_length) for _ in range(population_size)]

    for generation in range(generations):
        print(f"--- Generation {generation+1} ---")

        # 2. Calculate fitness and select parents
        parents = select_parents(population, population_size)

        # 3. Create the new generation
        next_generation = []
        for i in range(0, population_size, 2):
            parent1 = parents[i]
```

```
            parent2 = parents[i+1]

            # Crossover
            child1, child2 = crossover(parent1, parent2)

            # Mutation
            next_generation.append(mutate(child1, mutation_rate))
            next_generation.append(mutate(child2, mutation_rate))

        population = next_generation

        # Find the best chromosome in the current generation
        best_chromosome = max(population, key=calculate_fitness)
        best_fitness = calculate_fitness(best_chromosome)

        print(f"Best chromosome: {best_chromosome}")
        print(f"Best fitness: {best_fitness}")

        # 4. Termination condition
        if best_fitness == chromosome_length:
            print(f"Optimal solution found!")
            return best_chromosome

    # Return the best chromosome found after all generations
    return max(population, key=calculate_fitness)

# Parameters
CHROMOSOME_LENGTH = 16
POPULATION_SIZE = 100
GENERATIONS = 200
MUTATION_RATE = 0.01

# Run the algorithm
best_solution = genetic_algorithm(CHROMOSOME_LENGTH, POPULATION_SIZE,
GENERATIONS, MUTATION_RATE)
print("\n--- Final Results ---")
print("Final best solution:", best_solution)
print("Fitness:", calculate_fitness(best_solution))
```

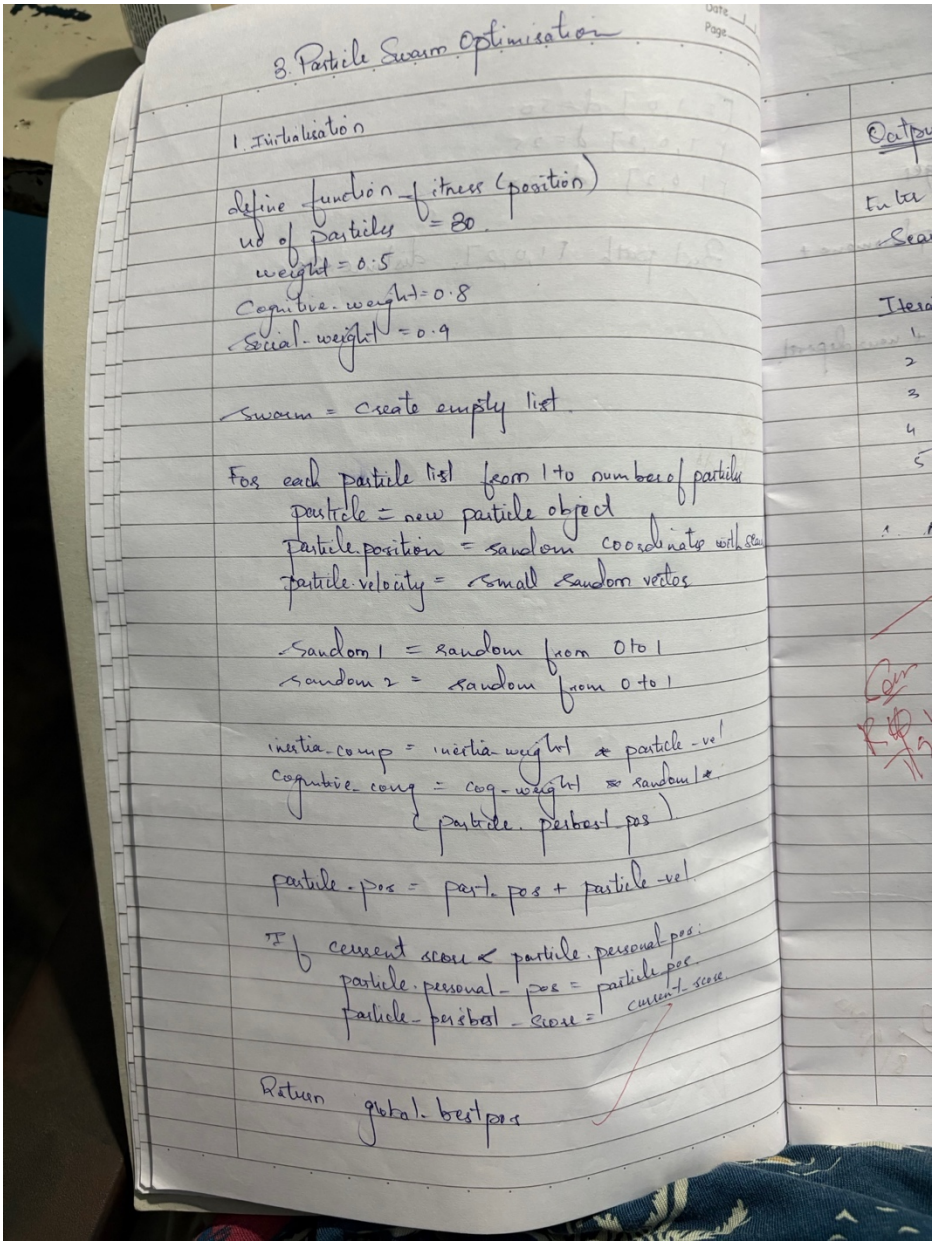
```
--- Generation 1 ---
Best chromosome: [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1]
Best fitness: 13
--- Generation 2 ---
Best chromosome: [1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]
Best fitness: 13
--- Generation 3 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1]
Best fitness: 14
--- Generation 4 ---
Best chromosome: [1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
Best fitness: 14
--- Generation 5 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
Best fitness: 15
--- Generation 6 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1]
Best fitness: 14
--- Generation 7 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
Best fitness: 15
--- Generation 8 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
Best fitness: 15
--- Generation 9 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
Best fitness: 15
--- Generation 10 ---
Best chromosome: [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Best fitness: 15
--- Generation 11 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
Best fitness: 15
--- Generation 12 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
Best fitness: 15
--- Generation 13 ---
Best chromosome: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Best fitness: 16
Optimal solution found!

--- Final Results ---
Final best solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Fitness: 16
```

Program 2

Particle Swarm Optimization for Function Optimization

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

Algorithm:



Date __/__/__
Page _____

Output.

Enter number of hikers = 10
Search iterations : 5

Iteration	Highest point found	max altitude
1	(1.3613, 1.8374)	98.2403
2	(1.5481, 1.9105)	98.6088
3	(1.7421, 2.4580)	98.6391
4	(2.2516, 3.0651)	98.9946
5	(2.959, 3.4771)	98.9948

1. Max altitude : 98.9940

Code:

```
import random

# Objective function: De Jong's Sphere function
def fitness(position):
    x, y = position
    return x**2 + y**2

# PSO parameters
num_particles = 10
max_iter = 10
w = 0.5       # inertia weight
c1 = 1.5      # cognitive coefficient
c2 = 1.5      # social coefficient

# Initialize particles (positions and velocities) in 2D space
particles = [[random.uniform(-10, 10), random.uniform(-10, 10)] for _ in range(num_particles)]
velocities = [[random.uniform(-1, 1), random.uniform(-1, 1)] for _ in range(num_particles)]

# Personal best positions and scores
pbest_positions = [p[:] for p in particles]
pbest_scores = [fitness(p) for p in particles]

# Global best position and score
gbest_index = pbest_scores.index(min(pbest_scores))
gbest_position = pbest_positions[gbest_index][:]
gbest_score = pbest_scores[gbest_index]

print(f"Initial global best position: {gbest_position}, fitness: {gbest_score:.6f}\n")

# Main PSO loop
for iteration in range(1, max_iter + 1):
    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]
```

```
        # Calculate fitness
        score = fitness(particles[i])

        # Update personal best
        if score < pbest_scores[i]:
            pbest_scores[i] = score
            pbest_positions[i] = particles[i][:]

            # Update global best
            if score < gbest_score:
                gbest_score = score
                gbest_position = particles[i][:]

    # Print iteration summary
    print(f"Iteration {iteration}: Global Best Position = {gbest_position}, Fitness = {gbest_score:.6f}")

# Final result
print("\nOptimization finished.")
print(f"Best position found: {gbest_position}")
print(f"Best fitness value: {gbest_score:.6f}")
```

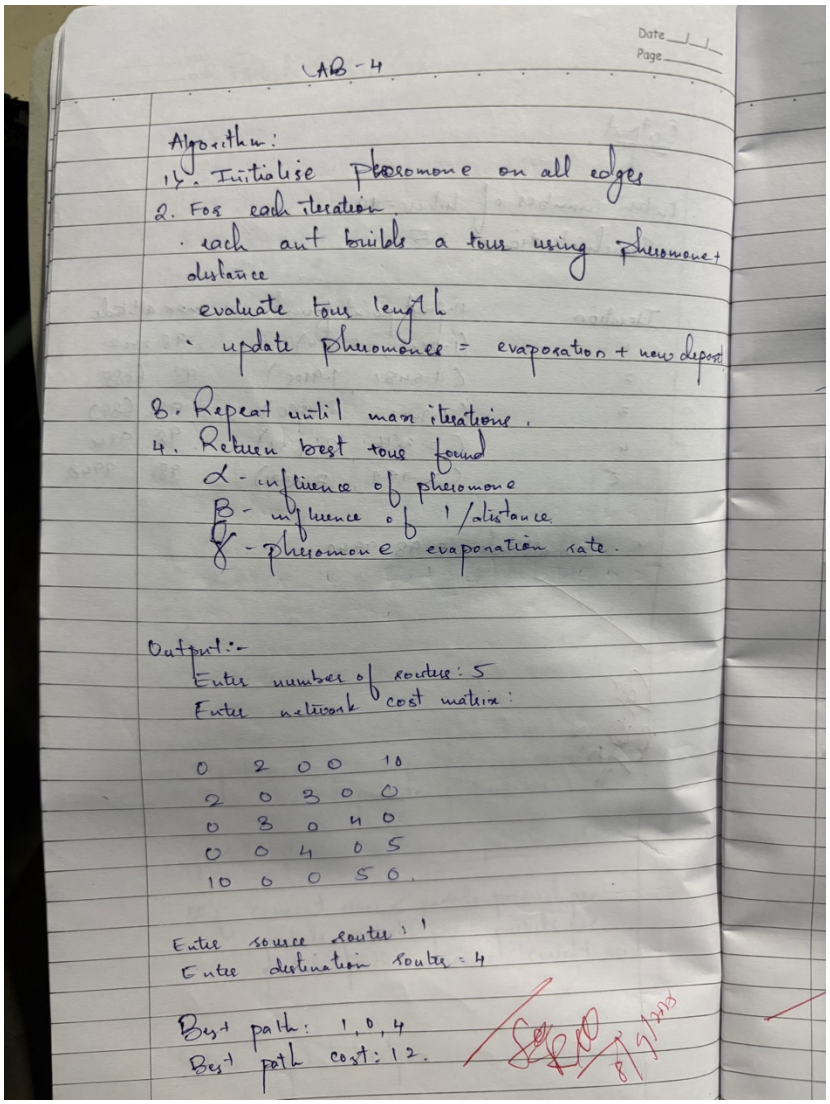
```
Initial global best position: [-3.454881045135405, 1.193311769389025], fitness: 13.360196

Iteration 1: Global Best Position = [-0.5581357017655004, -0.4775985964593339], Fitness = 0.539616
Iteration 2: Global Best Position = [-0.5581357017655004, -0.4775985964593339], Fitness = 0.539616
Iteration 3: Global Best Position = [-0.5111907134999516, 0.25867578014971526], Fitness = 0.328229
Iteration 4: Global Best Position = [-0.5111907134999516, 0.25867578014971526], Fitness = 0.328229
Iteration 5: Global Best Position = [-0.09720277420813872, 0.346786420138289], Fitness = 0.129709
Iteration 6: Global Best Position = [-0.19983824499450475, 0.02754817116790287], Fitness = 0.040694
Iteration 7: Global Best Position = [-0.10755422897951988, 0.04531099292834251], Fitness = 0.013621
Iteration 8: Global Best Position = [-0.10755422897951988, 0.04531099292834251], Fitness = 0.013621
Iteration 9: Global Best Position = [0.010400116262418768, 0.11380602298959472], Fitness = 0.013060
Iteration 10: Global Best Position = [0.04753920894840438, 0.006623855916628854], Fitness = 0.002304

Optimization finished.
Best position found: [0.04753920894840438, 0.006623855916628854]
Best fitness value: 0.002304
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

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

Algorithm:



LAB - 4

Date / /
Page

Algorithm:
1. Initialise pheromone on all edges
2. For each iteration
   - each ant builds a tour using pheromone + distance
   - evaluate tour length
   - update pheromones = evaporation + new deposit
3. Repeat until max iterations.
4. Return best tour found

α - influence of pheromone
β - influence of 1/distance
γ - pheromone evaporation rate.

Output:-
Enter number of routes: 5
Enter network cost matrix:

0 2 0 0 10
2 0 3 0 0
0 3 0 4 0
0 0 4 0 5
10 0 0 5 0

Enter source router: 1
Enter destination router = 4

Best path: 1, 0, 4
Best path cost: 12.

8/4/2025

Code:

```
import numpy as np
import random

class ACO_TSP:
    def __init__(self, dist_matrix, num_ants=10, alpha=1, beta=5, rho=0.5, Q=100, iterations=100):
        self.dist = dist_matrix
        self.n = len(dist_matrix)
        self.num_ants = num_ants
        self.alpha = alpha      # pheromone importance
        self.beta = beta        # heuristic importance
        self.rho = rho          # evaporation rate
        self.Q = Q              # pheromone deposit factor
        self.iterations = iterations

        # Initialize pheromone levels (small constant)
        self.pheromone = np.ones((self.n, self.n)) / self.n

        self.best_length = float('inf')
        self.best_path = None

    def run(self):
        for it in range(self.iterations):
            all_paths = []
            all_lengths = []

            # Each ant builds a tour
            for _ in range(self.num_ants):
                path = self.construct_solution()
                length = self.path_length(path)

                all_paths.append(path)
                all_lengths.append(length)

                # Update global best
                if length < self.best_length:
                    self.best_length = length
                    self.best_path = path

            # Update pheromone matrix
            self.update_pheromone(all_paths, all_lengths)
```

```
        print(f"Iteration {it+1}/{self.iterations} | Best length: {self.best_length}")

    return self.best_path, self.best_length

def construct_solution(self):
    path = []
    visited = set()
    current = random.randint(0, self.n - 1)
    path.append(current)
    visited.add(current)

    while len(visited) < self.n:
        probabilities = []
        for j in range(self.n):
            if j not in visited:
                tau = self.pheromone[current][j] ** self.alpha
                eta = (1 / self.dist[current][j]) ** self.beta
                probabilities.append(tau * eta)
            else:
                probabilities.append(0)

        probabilities = np.array(probabilities)
        probabilities /= probabilities.sum()  # normalize probabilities

        next_city = np.random.choice(range(self.n), p=probabilities)
        path.append(next_city)
        visited.add(next_city)
        current = next_city

    return path

def path_length(self, path):
    length = 0
    for i in range(len(path)):
        length += self.dist[path[i]][path[(i + 1) % self.n]]  # return to start
    return length

def update_pheromone(self, paths, lengths):
    # Evaporation
    self.pheromone *= (1 - self.rho)

    # Deposit pheromone
    for path, length in zip(paths, lengths):
```

```
            for i in range(len(path)):
                a, b = path[i], path[(i + 1) % self.n]
                self.pheromone[a][b] += self.Q / length
                self.pheromone[b][a] += self.Q / length  # symmetric TSP

# Example usage
if __name__ == "__main__":
    dist_matrix = np.array([
        [0, 2, 9, 10],
        [1, 0, 6, 4],
        [15, 7, 0, 8],
        [6, 3, 12, 0]
    ])

    aco = ACO_TSP(dist_matrix, num_ants=5, alpha=1, beta=5, rho=0.5, Q=100, iterations=30)
    best_path, best_length = aco.run()

    print("\nBest Path:", best_path)
    print("Best Length:", best_length)
```

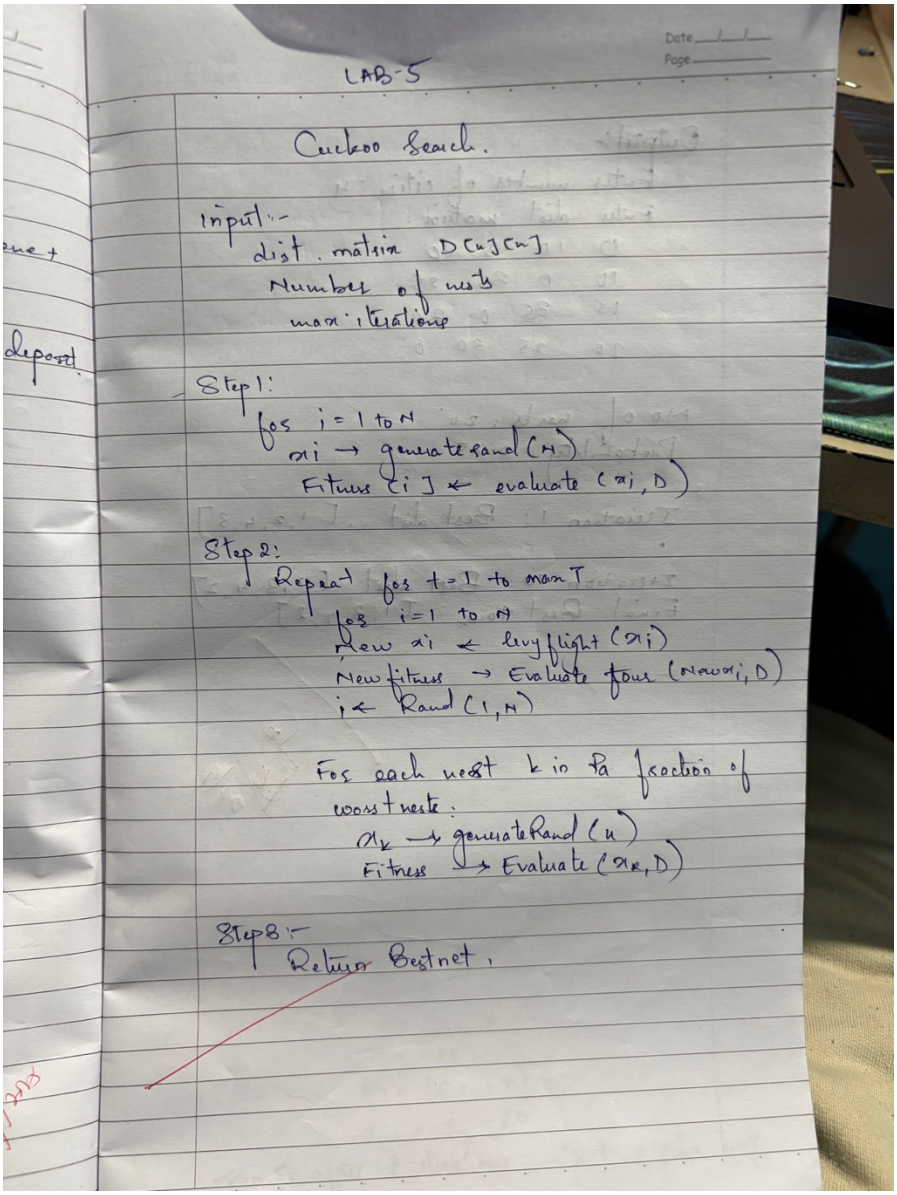
```
Iteration 1/30 | Best length: 21
Iteration 2/30 | Best length: 21
Iteration 3/30 | Best length: 21
Iteration 4/30 | Best length: 21
Iteration 5/30 | Best length: 21
Iteration 6/30 | Best length: 21
Iteration 7/30 | Best length: 21
Iteration 8/30 | Best length: 21
Iteration 9/30 | Best length: 21
Iteration 10/30 | Best length: 21
Iteration 11/30 | Best length: 21
Iteration 12/30 | Best length: 21
Iteration 13/30 | Best length: 21
Iteration 14/30 | Best length: 21
Iteration 15/30 | Best length: 21
Iteration 16/30 | Best length: 21
Iteration 17/30 | Best length: 21
Iteration 18/30 | Best length: 21
Iteration 19/30 | Best length: 21
Iteration 20/30 | Best length: 21
Iteration 21/30 | Best length: 21
Iteration 22/30 | Best length: 21
Iteration 23/30 | Best length: 21
Iteration 24/30 | Best length: 21
Iteration 25/30 | Best length: 21
...
Iteration 30/30 | Best length: 21

Best Path: [3, 1, 0, 2]
Best Length: 21
```

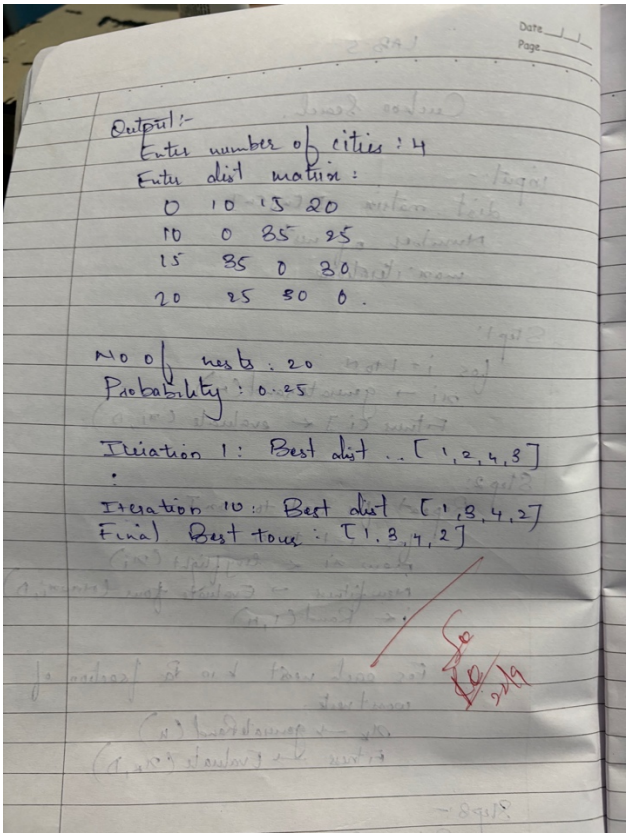
Program 4

Cuckoo Search (CS)

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

Algorithm:





Code:

```
import numpy as np
import random

# ---- Distance function ----
def route_length(route, dist_matrix):
    length = 0
    for i in range(len(route)):
        length += dist_matrix[route[i-1]][route[i]]
    return length

# ---- Lévy flight operator (approximation for discrete TSP) ----
def levy_flight(route):
    new_route = route.copy()
    # simple swap of two cities (local search)
    i, j = random.sample(range(len(route)), 2)
    new_route[i], new_route[j] = new_route[j], new_route[i]
    return new_route

# ---- Cuckoo Search Algorithm ----
def cuckoo_search_tsp(dist_matrix, n=15, pa=0.25, max_iter=500):
```

```
    num_cities = len(dist_matrix)

    # Step 1: Initialize nests (random routes)
    nests = [random.sample(range(num_cities), num_cities) for _ in range(n)]
    fitness = [route_length(r, dist_matrix) for r in nests]

    best_route = nests[np.argmin(fitness)]
    best_cost = min(fitness)

    for _ in range(max_iter):
        # Step 2: Generate new solutions via Lévy flight
        cuckoo = levy_flight(best_route)
        cuckoo_cost = route_length(cuckoo, dist_matrix)

        # Step 3: Replace a random nest if cuckoo is better
        j = random.randint(0, n-1)
        if cuckoo_cost < fitness[j]:
            nests[j] = cuckoo
            fitness[j] = cuckoo_cost

        # Step 4: Abandon fraction Pa of worst nests
        abandon_count = int(pa * n)
        worst_indices = np.argsort(fitness)[-abandon_count:]
        for idx in worst_indices:
            nests[idx] = random.sample(range(num_cities), num_cities)
            fitness[idx] = route_length(nests[idx], dist_matrix)

        # Step 5: Update global best
        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_cost:
            best_route = nests[current_best_idx]
            best_cost = fitness[current_best_idx]

    return best_route, best_cost

# ---- Example Usage ----
if __name__ == "__main__":
    # Example 5 cities distance matrix (symmetric TSP)
    dist_matrix = [
        [0, 2, 9, 10, 7],
        [2, 0, 6, 4, 3],
        [9, 6, 0, 8, 5],
        [10, 4, 8, 0, 6],
```

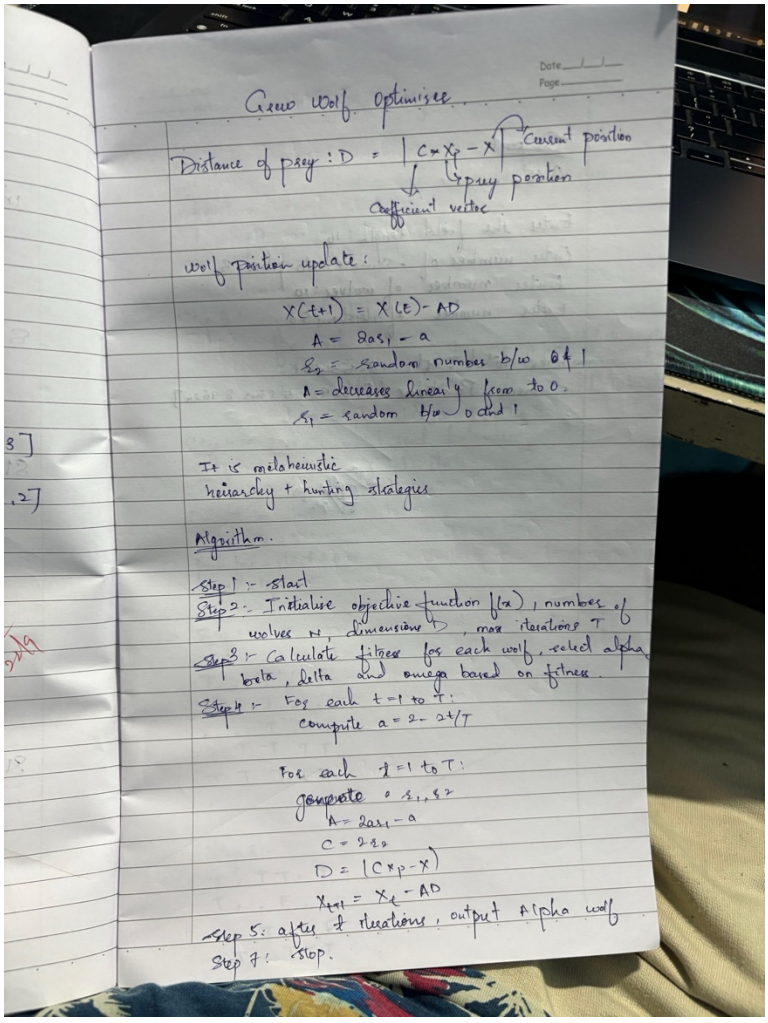
```
    [7, 3, 5, 6, 0]
]

best_route, best_cost = cuckoo_search_tsp(dist_matrix, n=20, pa=0.3, max_iter=1000)
print("Best Route:", best_route)
print("Best Cost:", best_cost)
```

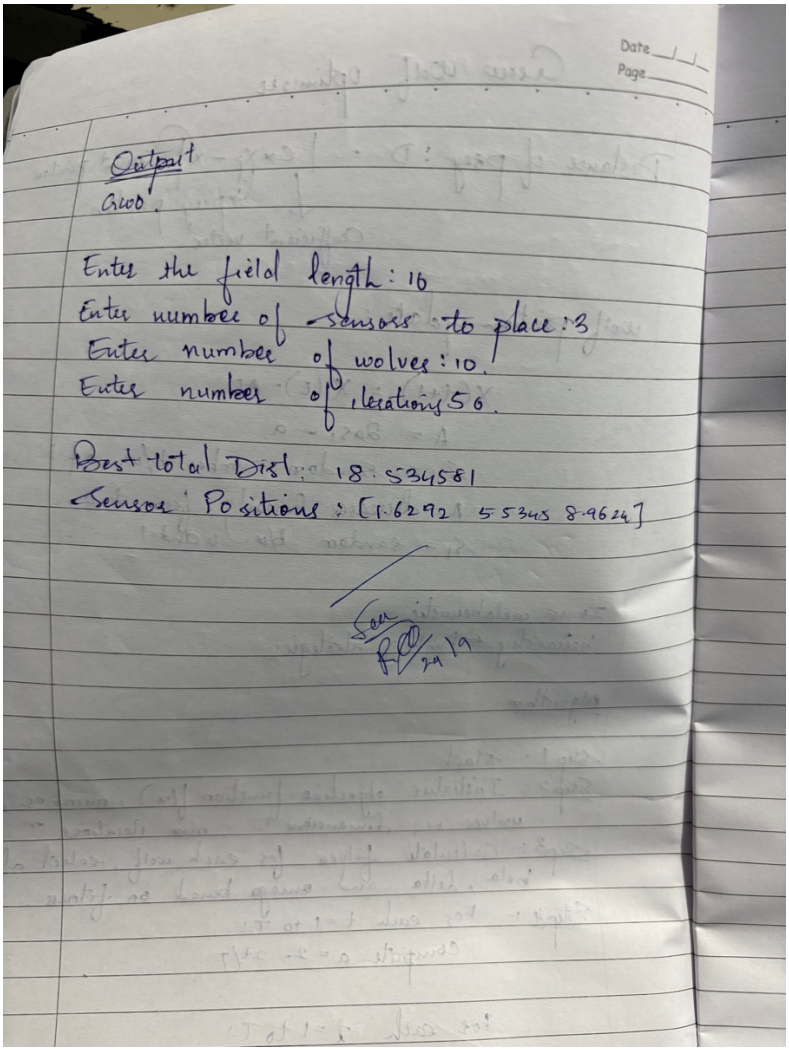
Program 5

Grey Wolf Optimizer (GWO)

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

Algorithm:





Code:

```
import numpy as np

# Knapsack fitness function
def fitness(solution, values, weights, capacity):
    total_value = np.sum(solution * values)
    total_weight = np.sum(solution * weights)
    if total_weight > capacity:
        return 0  # infeasible
    return total_value

# Repair function (greedy remove until feasible)
def repair(solution, values, weights, capacity):
    while np.sum(solution * weights) > capacity:
```

```
        idx = np.where(solution == 1)[0]
        if len(idx) == 0:
            break
        # remove least value/weight ratio item
        worst = min(idx, key=lambda i: values[i]/weights[i])
        solution[worst] = 0
    return solution

# Grey Wolf Optimizer for Knapsack
def GWO_knapsack(values, weights, capacity, n_wolves=10, max_iter=5):
    n_items = len(values)

    # Initialize wolves (random binary solutions)
    wolves = np.random.randint(0, 2, (n_wolves, n_items))
    for i in range(n_wolves):
        wolves[i] = repair(wolves[i], values, weights, capacity)

    # Evaluate fitness
    fitness_vals = [fitness(w, values, weights, capacity) for w in wolves]

    # Identify alpha, beta, delta
    sorted_idx = np.argsort(fitness_vals)[::-1]
    alpha = wolves[sorted_idx[0]].copy()
    beta  = wolves[sorted_idx[1]].copy()
    delta = wolves[sorted_idx[2]].copy()
    alpha_score = fitness_vals[sorted_idx[0]]

    # Main loop
    for t in range(max_iter):
        a = 2 - 2*(t/max_iter)  # linearly decreases from 2 to 0

        for i in range(n_wolves):
            X = wolves[i].copy()
            X_new = np.zeros(n_items)

            for d in range(n_items):
                r1, r2 = np.random.rand(), np.random.rand()
                A1 = 2*a*r1 - a
                C1 = 2*r2
                D_alpha = abs(C1*alpha[d] - X[d])
                X1 = alpha[d] - A1*D_alpha

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

```
            A2 = 2*a*r1 - a
            C2 = 2*r2
            D_beta = abs(C2*beta[d] - X[d])
            X2 = beta[d] - A2*D_beta

            r1, r2 = np.random.rand(), np.random.rand()
            A3 = 2*a*r1 - a
            C3 = 2*r2
            D_delta = abs(C3*delta[d] - X[d])
            X3 = delta[d] - A3*D_delta

            # Average position
            X_new[d] = (X1 + X2 + X3) / 3

            # Convert to binary (sigmoid + threshold)
            prob = 1 / (1 + np.exp(-X_new[d]))
            X_new[d] = 1 if np.random.rand() < prob else 0

        # Repair and evaluate
        X_new = repair(X_new.astype(int), values, weights, capacity)
        new_score = fitness(X_new, values, weights, capacity)

        # Replace if better
        if new_score > fitness_vals[i]:
            wolves[i] = X_new
            fitness_vals[i] = new_score

    # Update Alpha, Beta, Delta
    sorted_idx = np.argsort(fitness_vals)[::-1]
    alpha = wolves[sorted_idx[0]].copy()
    beta  = wolves[sorted_idx[1]].copy()
    delta = wolves[sorted_idx[2]].copy()
    alpha_score = fitness_vals[sorted_idx[0]]

    #  🔹 Print best solution of this iteration
    print(f"Iteration {t+1}: Best Solution = {alpha}, Best Value = {alpha_score}")

  return alpha, alpha_score


# Example usage
if __name__ == "__main__":
  values = np.array([15, 10, 9, 5, 8])
```

```
weights = np.array([1, 5, 3, 4, 2])
capacity = 8

best_solution, best_value = GWO_knapsack(values, weights, capacity, n_wolves=5, max_iter=8)
print("\nFinal Best Solution:", best_solution)
print("Final Best Total Value:", best_value)
```

```
Iteration 1: Best Solution = [1 0 1 0 1], Best Value = 32
Iteration 2: Best Solution = [1 0 1 0 1], Best Value = 32
Iteration 3: Best Solution = [1 0 1 0 1], Best Value = 32
Iteration 4: Best Solution = [1 1 0 0 1], Best Value = 33
Iteration 5: Best Solution = [1 1 0 0 1], Best Value = 33
Iteration 6: Best Solution = [1 1 0 0 1], Best Value = 33
Iteration 7: Best Solution = [1 1 0 0 1], Best Value = 33
Iteration 8: Best Solution = [1 1 0 0 1], Best Value = 33

Final Best Solution: [1 1 0 0 1]
Final Best Total Value: 33
```

Program 6

Parallel Cellular Algorithms and Programs

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

Algorithm:

Date / /
Page

Parallel Cellular Algo

Algo

Step 1:- Define problem and initialise parameters

Step 2 :- Evaluate fitness

fitness of cell = check if neighbouring is burning.

Step 3 :- Update states.

for each cell

-check neighbours ~~if burning, switch to empty~~ ~~loads of jobs~~ ~~assignments~~

~~overload~~ ~~conflict~~ True, then switch to burning

if empty, remains empty.

Step 4 :- Iterate until load balances

Step 5 :- Output final state.

6247

Output :-

Grid size : 5

prob_fire - start : 0.01

max - steps : 5.

Step 1 :

T . . T T F

F . T . .

. . T . .

T T T T T

T T T T T

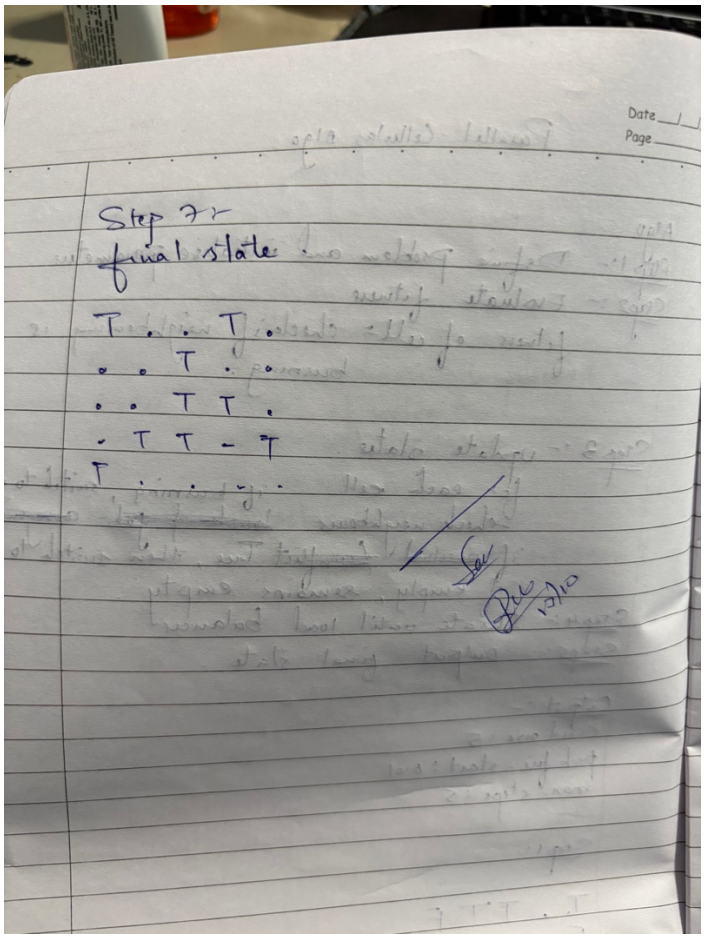
Step 5 :-

T . T T . T

T . . F . .

. . T . . F

. T . T T T



Code:

```
import random

# Fix random seed for reproducibility
random.seed(42)

# Job processing times for 6 jobs
jobs = [4, 7, 2, 6, 5, 3]  # total workload = 27
N = len(jobs)
M = 3  # number of machines

# Grid size (5x5 = 25 cells)
grid_size = (5, 5)
iterations = 5

# Moore neighborhood (including self)
neighbors_offsets = [(-1, -1), (-1, 0), (-1, 1),
                     (0, -1), (0, 0), (0, 1),
                     (1, -1), (1, 0), (1, 1)]
```

```
def initialize_population():
    population = []
    for _ in range(grid_size[0] * grid_size[1]):
        # Random assignment of jobs to machines
        assignment = [random.randint(0, M - 1) for _ in range(N)]
        population.append(assignment)
    return population

def calculate_makespan(assignment):
    loads = [0] * M
    for job_idx, machine in enumerate(assignment):
        loads[machine] += jobs[job_idx]
    return max(loads)

def get_neighbors_indices(idx):
    row = idx // grid_size[1]
    col = idx % grid_size[1]
    neighbors = []
    for dr, dc in neighbors_offsets:
        nr, nc = row + dr, col + dc
        if 0 <= nr < grid_size[0] and 0 <= nc < grid_size[1]:
            neighbors.append(nr * grid_size[1] + nc)
    return neighbors

def update_assignment(current, best_neighbor):
    # Copy one job assignment from best neighbor
    new_assignment = current[:]
    job_to_change = random.randint(0, N - 1)
    new_assignment[job_to_change] = best_neighbor[job_to_change]
    return new_assignment

# === PCA Execution ===

population = initialize_population()
best_solution = None
best_fitness = float('inf')

for iter in range(1, iterations + 1):
    fitness_values = [calculate_makespan(ind) for ind in population]

    # Track best
    min_fitness = min(fitness_values)
    if min_fitness < best_fitness:
```

```
    best_fitness = min_fitness
    best_solution = population[fitness_values.index(min_fitness)]

# Display current best
print(f"\nIteration {iter}")
print("Best Makespan:", best_fitness)
print("Best State: S =", best_solution)

# Update population
new_population = []
for idx, individual in enumerate(population):
    neighbors = get_neighbors_indices(idx)
    neighbor_fitness = [(calculate_makespan(population[n]), n) for n in neighbors]
    best_neighbor_idx = min(neighbor_fitness)[1]
    best_neighbor = population[best_neighbor_idx]
    updated_assignment = update_assignment(individual, best_neighbor)
    new_population.append(updated_assignment)

population = new_population
```

```
Iteration 1
Best Makespan: 10
Best State: S = [1, 2, 0, 1, 0, 2]

Iteration 2
Best Makespan: 10
Best State: S = [1, 2, 0, 1, 0, 2]

Iteration 3
Best Makespan: 10
Best State: S = [1, 2, 0, 1, 0, 2]

Iteration 4
Best Makespan: 9
Best State: S = [1, 2, 2, 0, 1, 0]

Iteration 5
Best Makespan: 9
Best State: S = [1, 2, 2, 0, 1, 0]
```

Program 7

Optimization via Gene Expression Algorithms

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

Algorithm:

LAB - 2.

Date / /
Page

Crossover algo.

1. Input no of cities
2. Input pop size.

For each generation:
    children = []
    for each pair (p1, p2) in population:
        child1 = crossover(p1, p2)
        child2 = crossover(p2, p1)
    for each child in children:
        randomly swap cities.
    for each route in population:
        distance = total tour length.
        fitness = -distance
END FOR.

Output:-

Population: [[0,1,2], [1,0,2], [2,0,1]]
after crossover: [[2,0,1], [2,1,0], [1,0,2]]
after mutation: [[2,1,0], [1,0,2], [1,0,2]]

Date __/__/__
Page

[2, 1, 0] d = 50.
[1, 0, 2] d = 25
[1, 0, 2] d = 25.

Best path [1, 0, 2], distance = 25

Sep 2018

Code:

```
import random
import math

#
================================================================================
======
# PARAMETERS
#
================================================================================
======
POP_SIZE = 150
GENE_LEN = 12
# We are only optimizing one variable: the launch angle (theta).
NUM_GENES = 1
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
NUM_GENS = 20
# Search space for the launch angle (theta) in radians.
# We search from 0 to pi/2 (0 to 90 degrees).
SEARCH_MIN = 0.0
SEARCH_MAX = math.pi / 2.0
CHROMOSOME_LEN = GENE_LEN * NUM_GENES

# Constants for the projectile motion equation
VELOCITY = 100.0  # Initial velocity in m/s (held constant)
GRAVITY = 9.81  # Acceleration due to gravity in m/s^2

#
================================================================================
======
# PROBLEM & FITNESS FUNCTION
#
================================================================================
======
def projectile_range(angle):
    """
```

```
    Calculates the horizontal range of a projectile.

    Formula: R = (v^2 * sin(2*theta)) / g

    Args:
        angle (float): The launch angle in radians.

    Returns:
        float: The horizontal range.
    """
    return (VELOCITY**2 * math.sin(2 * angle)) / GRAVITY

def express_gene(gene):
    """
    Translates a binary gene to a real-valued number (the launch angle).
    """
    decimal_val = int("".join(map(str, gene)), 2)
    max_decimal = (2**GENE_LEN) - 1
    return SEARCH_MIN + (decimal_val / max_decimal) * (SEARCH_MAX - SEARCH_MIN)

def express_chromosome(chromosome):
    """
    Translates a full chromosome into a solution vector.
    Since we only have one gene, this returns a single value.
    """
    return express_gene(chromosome[0:GENE_LEN])

def evaluate_fitness(pop):
    """
    Evaluates fitness for the population.
    Since we are maximizing the range, fitness is simply the range value.
    We add 1 to avoid a zero or negative fitness score.
    """
    return [projectile_range(express_chromosome(c)) + 1 for c in pop]

#
================================================================================
======
# GENETIC OPERATIONS
#
================================================================================
======
def initialize_population(size, length):
```

```
    """Generates a random initial population of binary chromosomes."""
    return [[random.randint(0, 1) for _ in range(length)] for _ in range(size)]

def select_parents(pop, fitness_scores):
    """Selects two parents using roulette wheel selection."""
    total_fitness = sum(fitness_scores)
    if total_fitness <= 0:
        return random.choice(pop), random.choice(pop)

    weights = [f / total_fitness for f in fitness_scores]
    return random.choices(pop, weights=weights, k=2)

def crossover(p1, p2, rate):
    """Performs single-point crossover."""
    if random.random() < rate:
        pt = random.randint(1, CHROMOSOME_LEN - 1)
        return p1[:pt] + p2[pt:], p2[:pt] + p1[pt:]
    return p1, p2

def mutate(c, rate):
    """Mutates a chromosome by flipping bits."""
    return [1 - bit if random.random() < rate else bit for bit in c]

#
================================================================================
======
# MAIN ALGORITHM LOOP
#
================================================================================
======
def run_gea():
    """Main function to run the Gene Expression Algorithm."""
    pop = initialize_population(POP_SIZE, CHROMOSOME_LEN)
    best_solution, best_value = None, -1.0

    print("Optimizing for Maximum Projectile Range...")

    for gen in range(NUM_GENS):
        fitness_scores = evaluate_fitness(pop)

        current_best_index = fitness_scores.index(max(fitness_scores))
        current_best_angle_rad = express_chromosome(pop[current_best_index])
        current_best_range = projectile_range(current_best_angle_rad)
```

```
        if current_best_range > best_value:
            best_value = current_best_range
            best_solution = current_best_angle_rad

        new_pop = []
        # Elitism: Keep the best solution from the current population
        best_chromosome = pop[current_best_index]
        new_pop.append(best_chromosome)

        for _ in range((POP_SIZE - 1) // 2):
            p1, p2 = select_parents(pop, fitness_scores)
            o1, o2 = crossover(p1, p2, CROSSOVER_RATE)
            new_pop.append(mutate(o1, MUTATION_RATE))
            new_pop.append(mutate(o2, MUTATION_RATE))

        pop = new_pop

        best_solution_deg = math.degrees(best_solution)
        print(f"Gen {gen + 1}/{NUM_GENS} - Best Range: {best_value:.4f} m (at
{best_solution_deg:.2f}°)")

    print("\n--- Optimization Complete ---")
    print(f"Final Best Angle: {math.degrees(best_solution):.2f}°")
    print(f"Final Best Range: {best_value:.4f} m")

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

```
Optimizing for Maximum Projectile Range...
Gen 1/20 - Best Range: 1019.2761 m (at 45.38°)
Gen 2/20 - Best Range: 1019.3211 m (at 45.27°)
Gen 3/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 4/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 5/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 6/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 7/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 8/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 9/20 - Best Range: 1019.3589 m (at 45.12°)
Gen 10/20 - Best Range: 1019.3661 m (at 45.05°)
Gen 11/20 - Best Range: 1019.3661 m (at 45.05°)
Gen 12/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 13/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 14/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 15/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 16/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 17/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 18/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 19/20 - Best Range: 1019.3673 m (at 45.03°)
Gen 20/20 - Best Range: 1019.3679 m (at 45.01°)

--- Optimization Complete ---
Final Best Angle: 45.01°
Final Best Range: 1019.3679 m
```