VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Dhruva S Rao
1BM23CS092

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 **Dhruva S Rao (1BM23CS092),** 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.

Mayanka Gupta Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

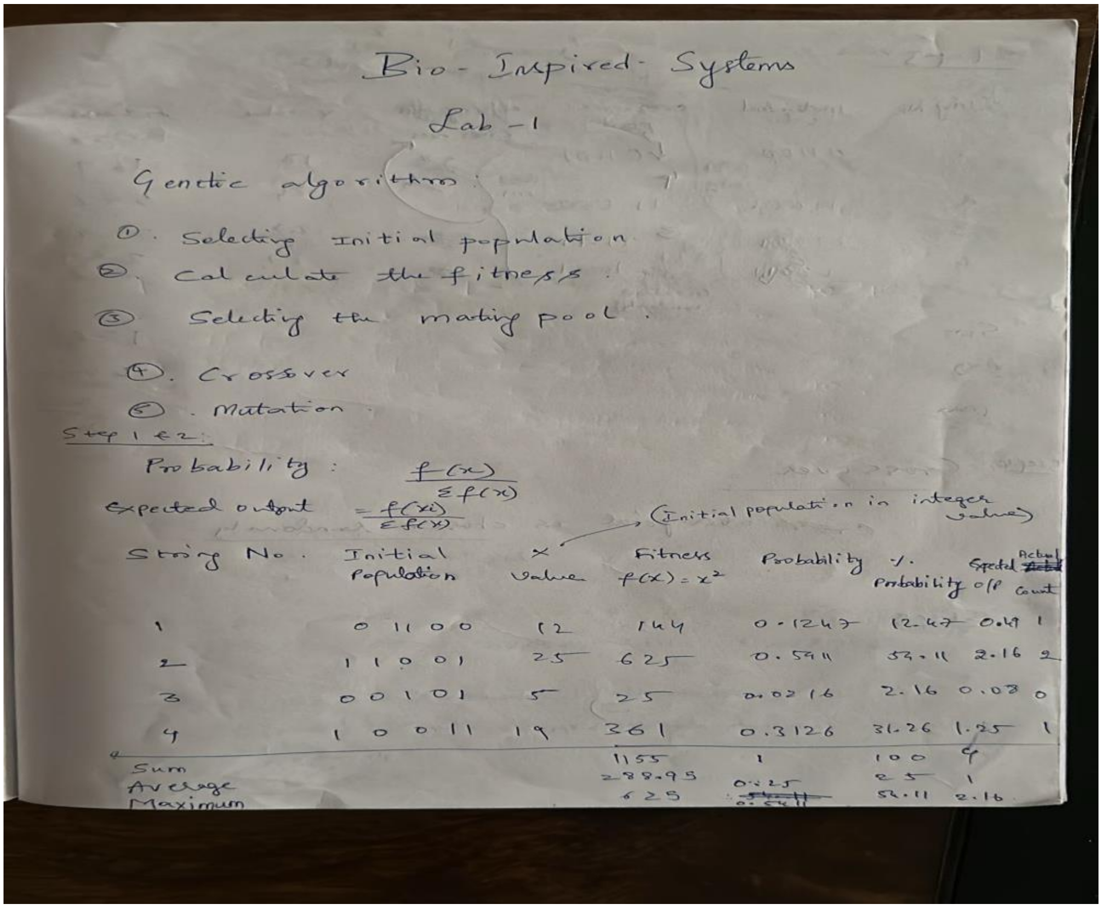
Index

Github Link: https://github.com/DhruvaSRao64/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:



Bio-Inspired Systems

Lab-1

Genetic algorithms:

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

Step 1 & 2:

Probability: $\frac{f(x)}{\Sigma f(x)}$

Expected output $= \frac{f(x_i)}{\Sigma f(x)}$

(Initial population in integer value)

String No.	Initial Population	x Value	Fitness $f(x) = x^2$	Probability	% Probability	Expected o/p	Actual Count
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	0.0216	2.16	0.08	0
4	10011	19	361	0.3126	31.26	1.25	1
Sum			1155	1	100	4	
Average			288.95	0.25	25	1	
Maximum			625	0.5411	54.11	2.16	

Step3:

Actual count 1

actual count 2

String No.	mating Pool	crossover point	offspring after cross over	x-value	fitness $f(x) = x^2$
1	01100		01101	13	169
2	11001	4	11000	24	576
3	11011	3	11011	27	729
4	10001		10001	17	289
Sum					1763
Avg.					440.75
Max.					729

4th bit mating

crossed over

2nd bit mating (we choose randomly)

actual count 1

Step 4: Crossover:

Cross over point is chosen randomly.

Step5:

=)

String No.	offspring after Crossover	Mutation Chromosome	offspring after mutation	X-value	Fitness $f(x) = x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	000000	11011	27	729
4	10001	00101	10100	20	400
					2546
Sum					636.5
Avg.					841
max.					

∵ there is one in last bit change 1st bit of change last bit

Lab1 — code and Output:

Initialize constants: POP_SIZE, GENES, MUTATION_RATE, CROSSOVER_RATE, GENERATIONS, X_RANGE

function Decode Chromosome (chromosome):
    Convert binary chromosome to decimal x in range X_RANGE
    Return x

```
Function fitness(x):
    Return x * sin(10 * π * x) + 1.0

Function genPopu():
    Return list of random binary chromosomes of length GE

Function EvaluatePopu(popu)
    For each individual in population:
        Decode chromosome
        Compute fitness
    Return list of fitness values

Function select(population, fitnesses):
    Use roulette wheel selection to pick one
    individual

Function crossover(p1, p2):
    if random() < CROSSOVER_RATE:
        Choose random crossover point
        Swap bits to create two children
    Else:
        Return parents unchanged
    Return child1, child2

Function mutate(chromosome):
    for each bit in chromosome:
        Flip bit with probability MUTATION_RATE
    Return mutated chromosome.

Procedure GeneticAlgo():
    population ← GenPopulation()
    all_best_gens ← empty list

    for gen = 1 to GENERATIONS:
        fitnesses ← EvaluatePopu(popu)
        new_population ← empty list

        Repeat (POP_SIZE/2) times
    population ← new_population
```

best_individual ← individual in population with highest fitness

Record (generation, best_individual, fitness) in all_best_gens

Sort all_best_gens by fitness descending

top_10 ← first 10 entries of sorted all_bests

for each entry in top_10:

Print generation number, decoded x, fitness

Print best solution found overall

Output:

Top 10 generations by Best fitness:

Gen 4 | x = 0.85158 | f(x) = 1.85053
Gen 2 | x = 0.85159 | f(x) = 1.85053
Gen 3 | x = 0.85159 | f(x) = 1.85053
Gen 1 | x = 0.85161 | f(x) = 1.85053
Gen 8 | x = 0.85158 | f(x) = 1.85053
Gen 9 | x = 0.85159 | f(x) = 1.85053
Gen 10 | x = 0.85159 | f(x) = 1.85053
Gen 14 | x = 0.85161 | f(x) = 1.85052
Gen 11 | x = 0.85185 | f(x) = 1.85043
Gen 12 | x = 0.85185 | f(x) = 1.85041

Best solution found overall:

$x = 0.85158$, $f(x) = 1.8503$

Application: ~~Engineering: opti~~ Biology: Simulating natural evolution (or) gene expression patterns

Code:

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

def fitness_function(x):
    return x * np.sin(10 * np.pi * x) + 1.0

POP_SIZE = 30
GENES = 16
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.7
GENERATIONS = 100

def generate_individual():
    return ''.join(random.choice('01') for _ in range(GENES))

def decode(individual):
    return int(individual, 2) / (2**GENES - 1)

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

def select(population, fitnesses):
    total_fit = sum(fitnesses)
    if total_fit == 0:
        return random.choices(population, k=2)
    probabilities = [f / total_fit for f in fitnesses]
    return random.choices(population, weights=probabilities, k=2)

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

def mutate(individual):
    return ''.join(
        bit if random.random() > MUTATION_RATE else random.choice('01')
        for bit in individual
    )

def genetic_algorithm():
    population = [generate_individual() for _ in range(POP_SIZE)]
    best_individual = population[0]
    best_fitness = fitness_function(decode(best_individual))
```

```
    fitness_history = []

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)
        max_fit = max(fitnesses)
        max_idx = fitnesses.index(max_fit)
        if max_fit > best_fitness:
            best_fitness = max_fit
            best_individual = population[max_idx]
        fitness_history.append(best_fitness)
        new_population = []
        while len(new_population) < POP_SIZE:
            parent1, parent2 = select(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population[:POP_SIZE]

    best_x = decode(best_individual)
    return best_x, best_fitness, fitness_history

best_x, best_fitness, history = genetic_algorithm()

print(f"Best solution found: x = {best_x:.5f}, f(x) = {best_fitness:.5f}")

plt.plot(history)
plt.title("Fitness over Generations")
plt.xlabel("Generation")
plt.ylabel("Best Fitness")
plt.grid(True)
plt.show()

import random

POP_SIZE = 100
NUM_CITIES = 20
GENERATIONS = 5
MUTATION_RATE = 5 / 100
CROSSOVER_RATE = 80 / 100

def generate_distance_matrix(num_cities):
    distance_matrix = [[0 if i == j else random.randint(10, 100) for j in range(num_cities)] for i in
range(num_cities)]
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            distance_matrix[j][i] = distance_matrix[i][j]
```

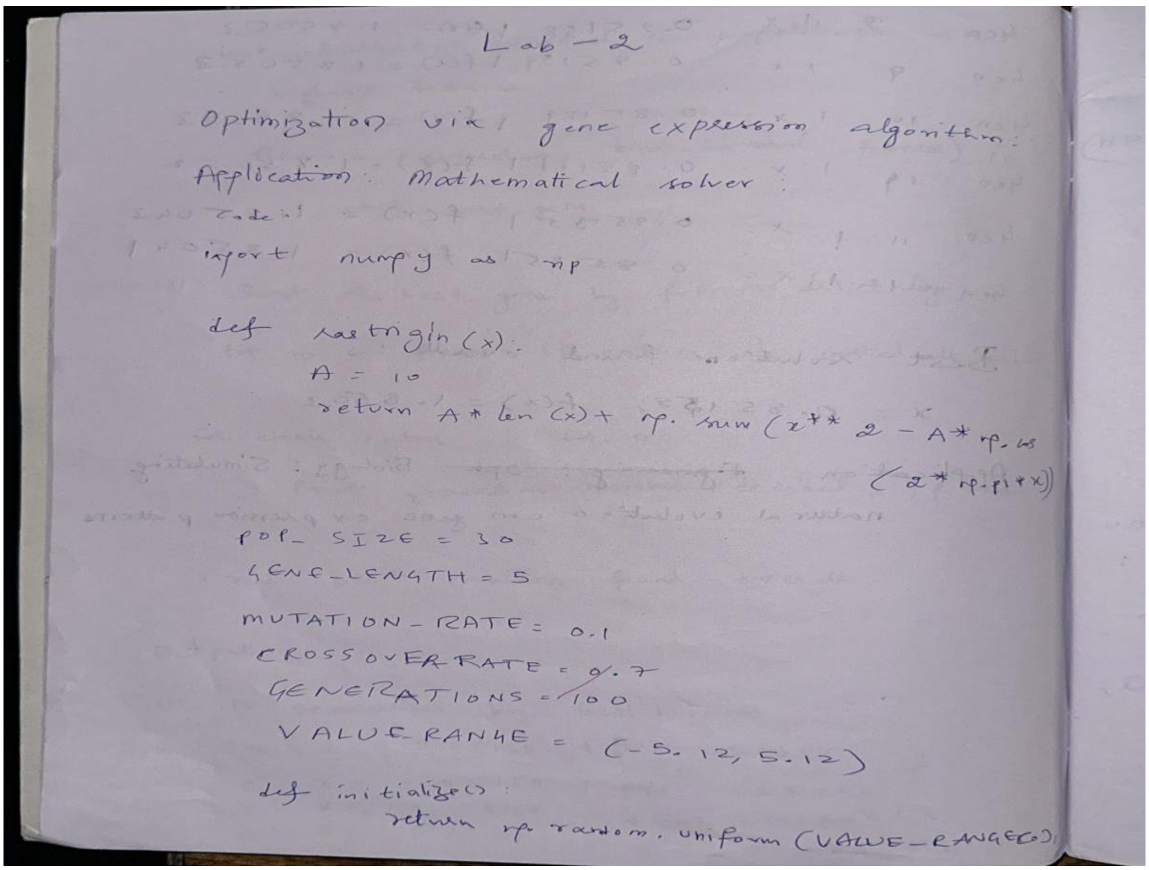
```
    return distance_matrix

DISTANCE_MATRIX = generate_distance_matrix(NUM_CITIES)

class Individual:
    def __init__(self):
        self.genome = random.sample(range(NUM_CITIES), NUM_CITIES)
        self.fitness = self.calculate_fitness()

    def calculate_fitness(self):
        total_distance = 0
        for i in range(NUM_CITIES - 1):
            total_distance += DISTANCE_MATRIX[self.genome[i]][self.genome[i + 1]]
        total_distance += DISTANCE_MATRIX[self.genome[NUM_CITIES - 1]][self.genome[0]]
        self.fitness = 1 / total_distance
        return self.fitness

    def mutate(self):
        if random.random() < MUTATION_RATE:
            i, j = random.sample(range(NUM_CITIES), 2)
            self.genome[i], self.genome[j] = self.genome[j], self.genome[i]
        self.fitness = self.calculate_fitness()

    @staticmethod
    def crossover(parent1, parent2):
        start, end = sorted(random.sample(range(NUM_CITIES), 2))
        child1_genome = [-1] * NUM_CITIES
        child2_genome = [-1] * NUM_CITIES
        child1_genome[start:end] = parent1.genome[start:end]
        child2_genome[start:end] = parent2.genome[start:end]
        fill_parent1 = [city for city in parent2.genome if city not in child1_genome]
        fill_parent2 = [city for city in parent1.genome if city not in child2_genome]
        for i in range(NUM_CITIES):
            if child1_genome[i] == -1:
                child1_genome[i] = fill_parent1.pop(0)
            if child2_genome[i] == -1:
                child2_genome[i] = fill_parent2.pop(0)
        child1 = Individual()
        child1.genome = child1_genome
        child1.fitness = child1.calculate_fitness()
        child2 = Individual()
        child2.genome = child2_genome
        child2.fitness = child2.calculate_fitness()
        return child1, child2

def selection(population):
    total_fitness = sum(individual.fitness for individual in population)
```

```
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual in population:
        current += individual.fitness
        if current > pick:
            return individual
    return population[-1]

def initialize_population():
    return [Individual() for _ in range(POP_SIZE)]

def best_individual(population):
    return min(population, key=lambda individual: 1 / individual.fitness)

def main():
    population = initialize_population()
    for generation in range(GENERATIONS):
        population.sort(key=lambda individual: individual.fitness, reverse=True)
        print(f"Generation {generation}: Best fitness = {population[0].fitness}, Distance = {1/population[0].fitness}")
        new_population = [population[0], population[1]]
        while len(new_population) < POP_SIZE:
            parent1 = selection(population)
            parent2 = selection(population)
            if random.random() < CROSSOVER_RATE:
                child1, child2 = Individual.crossover(parent1, parent2)
            else:
                child1, child2 = parent1, parent2
            child1.mutate()
            child2.mutate()
            new_population.append(child1)
            if len(new_population) < POP_SIZE:
                new_population.append(child2)
        population = new_population
    best_solution = best_individual(population)
    print("\nBest solution found:")
    print(f"Tour: {best_solution.genome}")
    print(f"Distance: {1 / best_solution.fitness}")

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

<u>Program 2</u>

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

Optimization via gene expression algorithm:

Application: Mathematical solver:

```
import numpy as np

def rastrigin(x):
    A = 10
    return A * len(x) + np.sum(x** 2 - A* np.cos
                               (2 * np.pi * x))

POP_SIZE = 30
GENE_LENGTH = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
GENERATIONS = 100
VALUE_RANGE = (-5.12, 5.12)

def initialize():
    return np.random.uniform(VALUE_RANGE[0],
```

```
        VALUE_RANGE[1], (POP_SIZE, GENE_LENGTH))

def evaluate(pop):
    return np.array([rastrigin(ind) for ind in
                    pop])

def select(pop, fitness, k=3):
    selected = []
    for _ in range(len(pop)):
        idx = np.random.choice(len(pop), k)
        winner = pop[idx[np.argmin(fitness[idx])]]
        selected.append(winner)
    return np.array(selected)

def crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        p1, p2 = parents[i], parents[(i+1) %
                                      len(parents)]
```

```
if np.random.rand() < CROSSOVER_RATE:
    point = np.random.randint(1, GENE_LENGTH)
    c1 = np.concatenate((p1[:point],
                          p2[point:]))
    c2 = np.concatenate((p2[:point],
                          p1[point:]))
def mutate(pop):
    for individual in pop:
        for i in range(GENE_LENGTH):
            if np.random.rand() < MUTATION_
                                  RATE:
                individual[i] = np.random.
                                 uniform
                    (VALUE_RANGE[0],
                     VALUE_RANGE[1])
    return pop
```

```
best, fitness = gene_expression_algorithm()
print("\n Best Solution:", best)
print("Best Fitness:", fitness)
```

Output:

Generation 0: Best Fitness = 44.2829
Generation 1: Best Fitness = 44.2829
Generation 2: Best Fitness = 44.2829
Generation 3: Best Fitness = 31.2741 Apply not Application
Generation 4: Best Fitness = 24.2802
Generation 5: Best Fitness = 22.1072
Generation 6: Best Fitness = 17.5759
Generation 7: Best Fitness = 17.5759
Generation 8: Best Fitness = 16.7390
Generation 9: Best Fitness = 16.7390
25/8/2025 Generation 10: Best Fitness = 16.7390

Code:

```
import random

POP_SIZE = 20
GENES = 5
GENERATIONS = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7

def fitness_function(treatment_plan):
    survival_rate = sum(treatment_plan) / len(treatment_plan)
    return survival_rate

class Individual:
    def __init__(self):
        self.genome = [random.randint(0, 10) for _ in range(GENES)]
        self.fitness = self.calculate_fitness()

    def calculate_fitness(self):
        return fitness_function(self.genome)
```

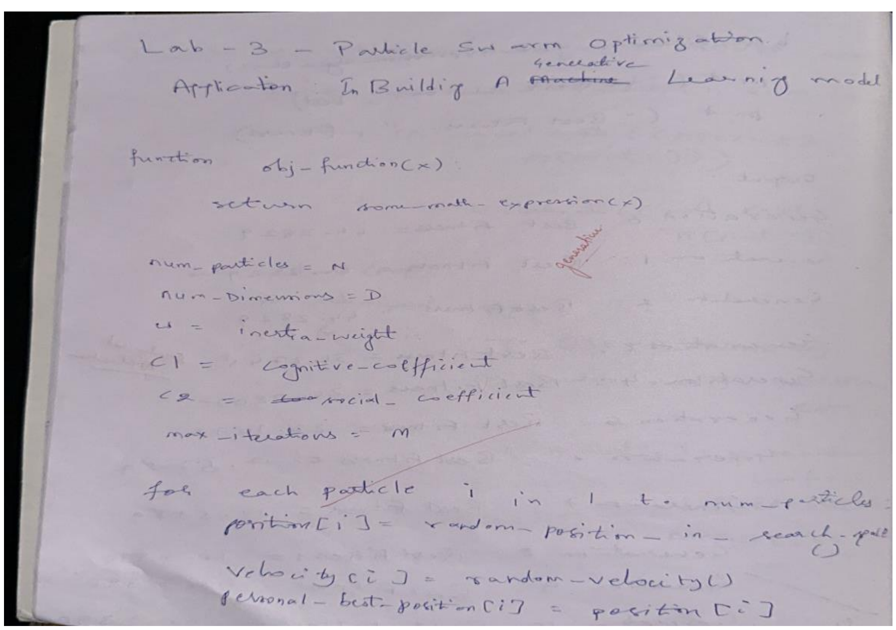
```
    def mutate(self):
        if random.random() < MUTATION_RATE:
            gene_idx = random.randint(0, GENES - 1)
            self.genome[gene_idx] = random.randint(0, 10)
        self.fitness = self.calculate_fitness()

    @staticmethod
    def crossover(parent1, parent2):
        crossover_point = random.randint(1, GENES - 1)
        child1_genome = parent1.genome[:crossover_point] + parent2.genome[crossover_point:]
        child2_genome = parent2.genome[:crossover_point] + parent1.genome[crossover_point:]
        child1 = Individual()
        child1.genome = child1_genome
        child1.fitness = child1.calculate_fitness()
        child2 = Individual()
        child2.genome = child2_genome
        child2.fitness = child2.calculate_fitness()
        return child1, child2

def selection(population):
    total_fitness = sum(individual.fitness for individual in population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual in population:
        current += individual.fitness
        if current > pick:
            return individual
    return population[-1]

def initialize_population():
    return [Individual() for _ in range(POP_SIZE)]

def best_individual(population):
    return max(population, key=lambda individual: individual.fitness)

def main():
    population = initialize_population()
    for generation in range(GENERATIONS):
        population.sort(key=lambda individual: individual.fitness, reverse=True)
        print(f"Generation {generation}: Best fitness = {population[0].fitness}, Genome =
{population[0].genome}")
        new_population = [population[0], population[1]]
        while len(new_population) < POP_SIZE:
            parent1 = selection(population)
            parent2 = selection(population)
            if random.random() < CROSSOVER_RATE:
                child1, child2 = Individual.crossover(parent1, parent2)
```

```
        else:
            child1, child2 = parent1, parent2
        child1.mutate()
        child2.mutate()
        new_population.append(child1)
        if len(new_population) < POP_SIZE:
            new_population.append(child2)
    population = new_population
  best = best_individual(population)
  print("\nBest treatment plan found:")
  print(f"Genome: {best.genome}, Fitness (Survival Rate): {best.fitness}")

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

Program 3

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.

Algorithm:


Lab - 3 - Particle Swarm Optimization
Application In Building A ~~Machine~~ Generative Learning model

function obj_function(x)
    return some_math_expression(x)

num_particles = N
num_Dimensions = D
w = inertia_weight
C1 = Cognitive_coefficient
C2 = social_coefficient
max_iterations = M

for each particle i in 1 to num_particles
    position[i] = random_position_in_search_space()
    Velocity[i] = random_velocity()
    personal_best_position[i] = position[i]

```
personal_best_fitness [i] = objective_function(position[i])

global_best_position = particle with best personal_best_
                       fitness

global_best_fitness = best fitness value

for iteration in 1 to max_iterations
    for each particle i in 1 to num_particles:
        fitness = obj_func (position[i])

        if fitness < personal_best_fitness[i]:
            personal_best_fitness[i] = fitness
            personal_best_position[i] = position[i]

    best
    for each particle i in 1 to num_particles:
        for each dimension d in 1 to num_dim
            r1 = random()   r2 = random()
            velocity[i][d] = (w * velocity[i][d])
                + (c1 * r1 * (person_best[i][d] -
                                position[i][d]))

            position[i][d] += velocity[i][d]

print(global_best_position)
print(global_best_fitness)
```

Iteration 1/5 - Best Fitness : 9.41627
Iteration 2/5 - Best Fitness : 3.38322
Iteration 3/5 - Best Fitness : 3.38322
Iteration 4/5 - Best Fitness : 3.32767
Iteration 5/5 - Best Fitness : 1.82257
Best_position : [4.772114 , 0.3793207]
Best Fitness value : 22.916956

MG
1 Sept

Code:

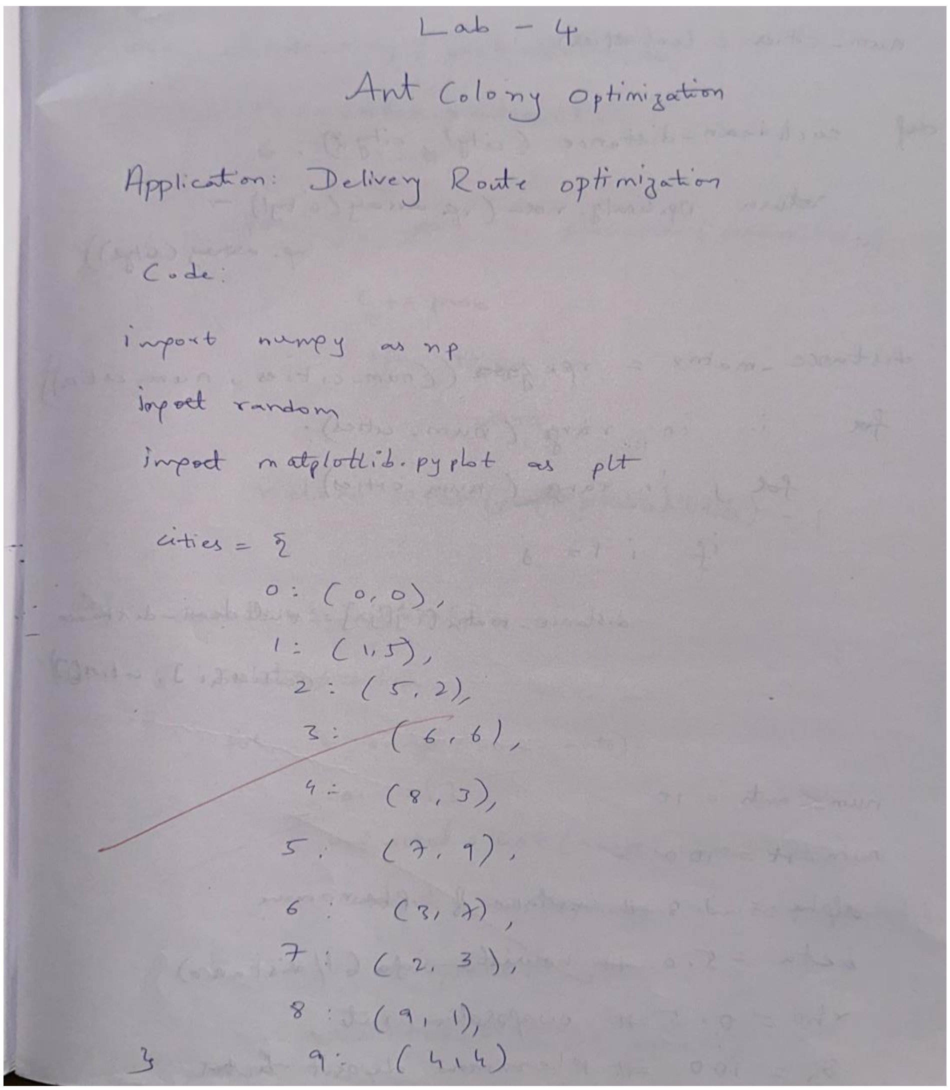
```
import random
import math
start = (0, 0)
goal = (10, 10)
obstacles = [((3, 3), (5, 5)), ((6, 7), (8, 9))]
def is_collision(p1, p2):
    for (bl, tr) in obstacles:
        x1, y1 = bl
        x2, y2 = tr
        if (min(p1[0], p2[0]) < x2 and max(p1[0], p2[0]) > x1 and
            min(p1[1], p2[1]) < y2 and max(p1[1], p2[1]) > y1):
            return True
    return False
def path_length(path):
    length = 0
    for i in range(len(path)-1):
        p1, p2 = path[i], path[i+1]
        if is_collision(p1, p2):
            return 10**6
        length += math.dist(p1, p2)
    return length

class Particle:
    def __init__(self, num_waypoints, bounds):
        self.position = [(random.uniform(bounds[0][0], bounds[0][1]),
                          random.uniform(bounds[1][0], bounds[1][1]))
                         for _ in range(num_waypoints)]
        self.velocity = [(0, 0) for _ in range(num_waypoints)]
        self.best_position = list(self.position)
        self.best_value = float("inf")
    def evaluate(self, func):
        path = [start] + self.position + [goal]
        value = func(path)
        if value < self.best_value:
            self.best_value = value
            self.best_position = list(self.position)
        return value

    def update_velocity(self, global_best, w, c1, c2):
        new_velocity = []
        for i in range(len(self.position)):
            r1, r2 = random.random(), random.random()
            vx = (w * self.velocity[i][0] +
                  c1 * r1 * (self.best_position[i][0] - self.position[i][0]) +
                  c2 * r2 * (global_best[i][0] - self.position[i][0]))
```

```
            vy = (w * self.velocity[i][1] +
                  c1 * r1 * (self.best_position[i][1] - self.position[i][1]) +
                  c2 * r2 * (global_best[i][1] - self.position[i][1]))
            new_velocity.append((vx, vy))
        self.velocity = new_velocity

    def update_position(self, bounds):
        new_position = []
        for i in range(len(self.position)):
            x = self.position[i][0] + self.velocity[i][0]
            y = self.position[i][1] + self.velocity[i][1]
            x = max(bounds[0][0], min(x, bounds[0][1]))
            y = max(bounds[1][0], min(y, bounds[1][1]))
            new_position.append((x, y))
        self.position = new_position

class PSO:
    def __init__(self, func, num_waypoints=3, bounds=[(0, 10), (0, 10)],
                 num_particles=20, max_iter=100, w=0.5, c1=1.5, c2=1.5):
        self.func = func
        self.num_waypoints = num_waypoints
        self.bounds = bounds
        self.swarm = [Particle(num_waypoints, bounds) for _ in range(num_particles)]
        self.global_best_position = list(self.swarm[0].position)
        self.global_best_value = float("inf")
        self.max_iter = max_iter
        self.w, self.c1, self.c2 = w, c1, c2

    def run(self):
        for _ in range(self.max_iter):
            for particle in self.swarm:
                value = particle.evaluate(self.func)
                if value < self.global_best_value:
                    self.global_best_value = value
                    self.global_best_position = list(particle.best_position)
            for particle in self.swarm:
                particle.update_velocity(self.global_best_position, self.w, self.c1, self.c2)
                particle.update_position(self.bounds)
        return self.global_best_position, self.global_best_value
if __name__ == "__main__":
    pso = PSO(func=path_length, num_waypoints=3, max_iter=100)
    best_path, best_value = pso.run()
    full_path = [start] + best_path + [goal]
    print("Best Path Found:")
    for p in full_path:
        print(p)
    print("Total Path Length:", best_value”)
```

Program 4

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

Ant Colony Optimization

Application: Delivery Route optimization

Code:

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

cities = {
        0: (0, 0),
        1: (1, 5),
        2: (5, 2),
        3: (6, 6),
        4: (8, 3),
        5: (7, 9),
        6: (3, 7),
        7: (2, 3),
        8: (9, 1),
        9: (4, 4)
}
```

```
num_cities = len(cities)

def euclidean_distance (city1, city2):
        return np.linalg.norm (np.array(city1) -
                                        np.array(city2))

distance_matrix = np.zeros ((num_cities , num_cities))
for i in range (num_cities):
        for j in range (num_cities):
                if i != j :
                        distance_matrix[i][j] = euclidean_distance
                                        (cities[i], cities[j])

num_ants = 10
num_it = 100
alpha = 1.0 # importance of pheromone
beta = 5.0 # heuristic info (1/distance)
rho = 0.5 # evaporation rate
Q = 100 # pheromone deposit factor
```

```
def choose_city (probabilities).
    r = random.random ()
    c = 0.0
    for i, prob in enumerate ( probabilities).
        c += prob
        if r <= c
            return i

    return len (probabilities) - 1

def construct_solution ():
    solutions = []
    for _ in range (num_ants):
        tour = []
        unvisited = list (range (num_cities))
        current = random.choice (unvisited)
        tour.append (current)
        unvisited.remove(current)
        while unvisited:
            probabilities = []
            for next_city in unvisited:
                tau = pheromone [current][next_city
                eta = (1/ distance_matrix[current]
                       [next_city]) ** beta

    return solutions

def calculate_tour_length ( tour).
    return sum ( distance_matrix (tour [i]] [
                 tour [(i+1) % num_cities]]
           for i in range ( num_cities))
best_tour = None
best_length = float ('inf')
for it in range (num_it):
    solutions = construct_solution()
```

```
for tour in solutions:
        length = calculate_tour_length(tour)
        if      length < best_length:
                best_length = length
                best_tour   = tour

print("\n Best Tour: best_tour)
print("Best Tour length: ", best_length)
def   plot_tour(tour):
      x = [ cities[i][0] for  i  in tour] + [cities[tour[0]
                                              ][0]]
      y = [cities[i][1] for  i in tour] + cities[tour[0]][1]]

      plt.figure(figsize = (10, 6))
      plt.plot( x, y, 'o-')
      plt.show()
plt_tour(best_tour)
```

Output

Iteration 1/5 - Best length: 39.17

Iteration 2/5 - Best length: 37.68

Iteration 3/5 - Best length: 37.22

Iteration 4/5 - Best length: 37.22

Iteration 5/5 - Best length: 37.22

Best Tour found: [0, 7, 1, 6, 3, 5, 8, 4, 9, 2]

Tour Length: 37.22121933703379 4

MG
8/9/25

Code:

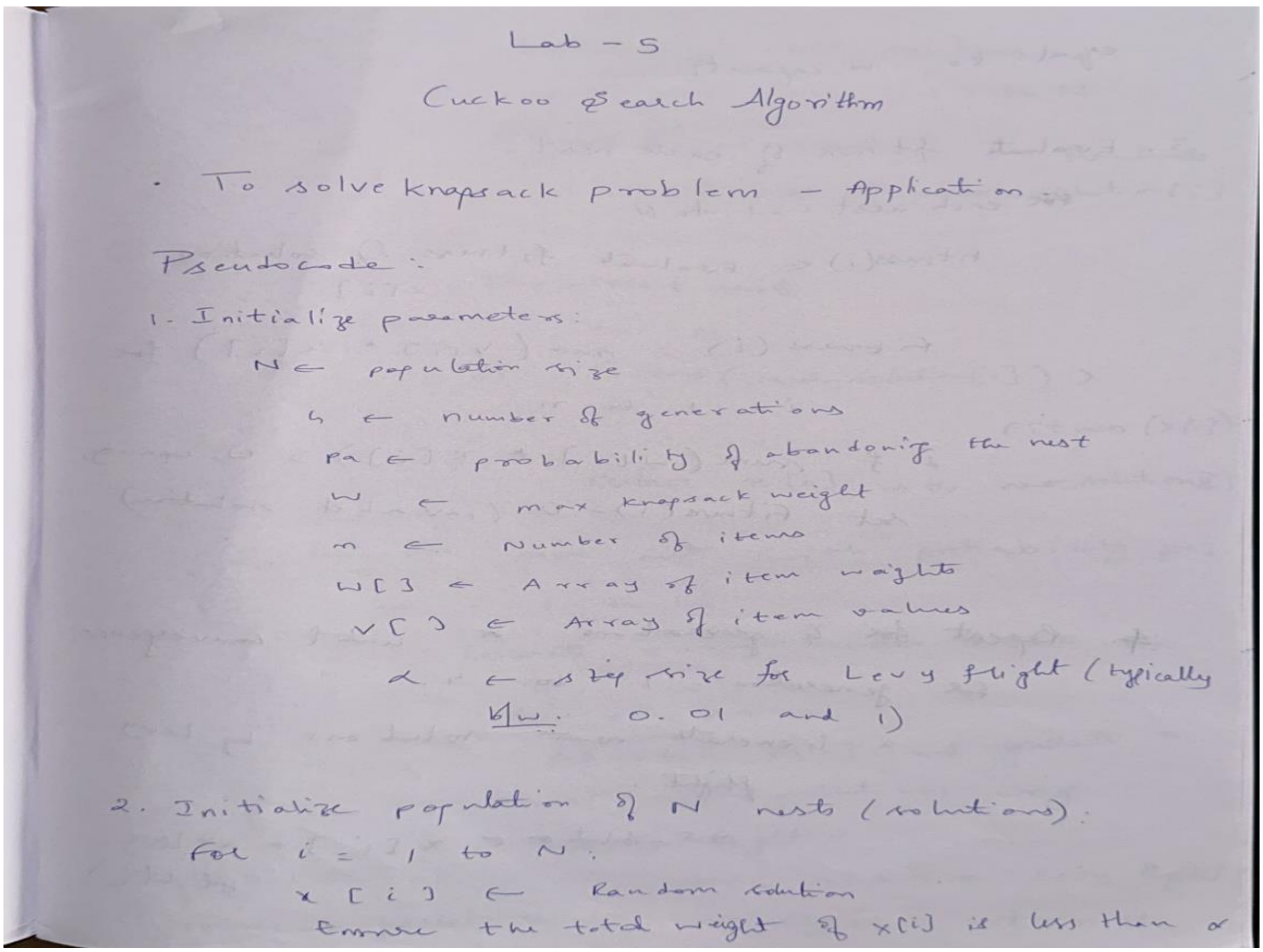
```
import random
import numpy as np

def calculate_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def ant_colony_optimization(cities, n_ants, n_best, n_iterations, decay, alpha=1, beta=5, Q=100):
    n_cities = len(cities)
    dist = np.zeros((n_cities, n_cities))
    for i in range(n_cities):
        for j in range(n_cities):
            dist[i][j] = calculate_distance(cities[i], cities[j])
    pheromone = np.ones((n_cities, n_cities)) * 0.1
    best_path = None
    best_path_length = float('inf')
    for _ in range(n_iterations):
        all_paths = []
        all_lengths = []
        for ant in range(n_ants):
            path = []
            visited = [False] * n_cities
            current_city = random.randint(0, n_cities - 1)
            path.append(current_city)
            visited[current_city] = True
            for _ in range(n_cities - 1):
                next_city = choose_next_city(current_city, visited, pheromone, dist, alpha, beta)
                path.append(next_city)
                visited[next_city] = True
                current_city = next_city
            path.append(path[0])
            path_length = calculate_path_length(path, dist)
            all_paths.append(path)
            all_lengths.append(path_length)
            if path_length < best_path_length:
                best_path_length = path_length
                best_path = path
        pheromone *= (1 - decay)
        for path, length in zip(all_paths[:n_best], all_lengths[:n_best]):
            for i in range(len(path) - 1):
                pheromone[path[i]][path[i+1]] += Q / length
        print(f"Best path length so far: {best_path_length}")
    return best_path, best_path_length

def choose_next_city(current_city, visited, pheromone, dist, alpha, beta):
```

```
    n_cities = len(pheromone)
    probabilities = []
    for i in range(n_cities):
        if not visited[i]:
            pheromone_level = pheromone[current_city][i] ** alpha
            distance_factor = (1.0 / dist[current_city][i]) ** beta
            probabilities.append(pheromone_level * distance_factor)
        else:
            probabilities.append(0)
    total_prob = sum(probabilities)
    probabilities = [p / total_prob for p in probabilities]
    next_city = random.choices(range(n_cities), weights=probabilities)[0]
    return next_city

def calculate_path_length(path, dist):
    length = 0
    for i in range(len(path) - 1):
        length += dist[path[i]][path[i+1]]
    return length

if __name__ == "__main__":
    cities = [
        (0, 0),
        (1, 2),
        (2, 4),
        (3, 1),
        (5, 0),
        (6, 3)
    ]
    n_ants = 10
    n_best = 5
    n_iterations = 100
    decay = 0.95
    best_path, best_path_length = ant_colony_optimization(cities, n_ants, n_best, n_iterations, decay)
    print("Best path found:", best_path)
    print("Length of best path:", best_path_length)
```

<u>Program 5</u>

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:



Lab - 5

Cuckoo Search Algorithm

• To solve knapsack problem - Application.

Pseudocode:

1. Initialize parameters:
   N ← population size
   G ← number of generations
   Pa ← probability of abandoning the nest
   W ← max knapsack weight
   n ← Number of items
   w[] ← Array of item weights
   v[] ← Array of item values
   α ← step size for Levy flight (typically b/w. 0.01 and 1)

2. Initialize population of N nests (solutions):
   For i = 1 to N:
   x[i] ← Random solution
   Ensure the total weight of x[i] is less than or

equal to W_capacity.

3. Evaluate fitness of each nest:

for each nest i = 1 to N:

fitness(i) ← Evaluate fitness of solution x[i]

fitness(i) = sum(v[i] * x[i]) for i = 1 to n

If sum(w[i] * x[i]) > W_capacity, set fitness(i) = 0 (invalid solution)

4. Repeat for G generations or until convergence:

for generation = 1 to G:

a. Generate new solutions by Levy flight:

new_solution ← x[i] + α * Levy_flight()

b. Evaluate fitness of new solutions:
for each new nest i = 1 to N:
Fitness (new_solution) ← Evaluate fitness of new_solution[i]

c. Replace the worst nests:
for i = 1 to N:
If Fitness (new_solution [i]) > Fitness (x[i]):
Replace x [i] with new_solution[i]

d. Abandon some nests with probability pa:
for i = 1 to N:
Generate random number r ∈ [0 . 1]
If r < pa:
Abandon x[i] and generate a new nest (solution) using Levy flight:
new_solution ← Levy_flight()

5. After G generations (or) convergence, return the best solution:
Best_nest ← the nest with the highest fitness value.
Return Best_nest as the solution to the knapsack problem

6. End

Output:
Top 5 solutions (in terms of fitness):
Solution 1: [1 1 0 1 1], Fitness: 13
Solution 2: [1 1 0 1 1], fitness: 13
Solution 3: [1 1 0 1 1], Fitness: 13
Solution 4: [1 1 0 1 1], Fitness: 13
Solution 5: [1 1 0 1 1], Fitness: 13

Code:

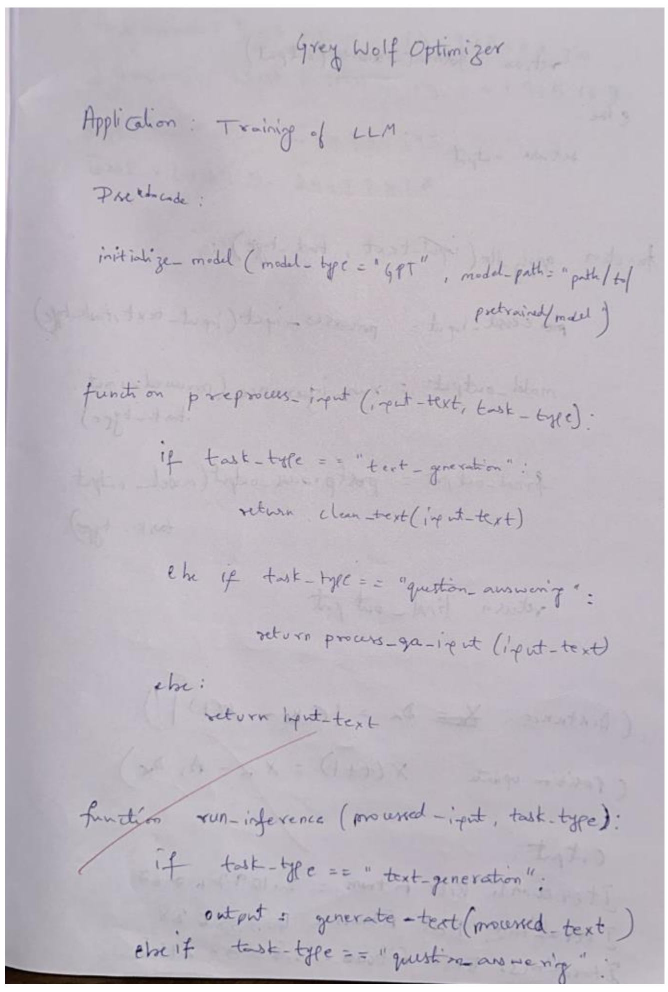
```
import random
import math

def distance(a, b):
    return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)

def tour_length(tour, cities):
    total = 0.0
    n = len(tour)
    for i in range(n - 1):
        total += distance(cities[tour[i]], cities[tour[i + 1]])
    total += distance(cities[tour[-1]], cities[tour[0]])
    return total

def levy_step_length(beta=1.5):
    u = random.random()
    step = int(1 / (u ** (1 / beta)))
    return max(1, step)

def discrete_levy_flight(tour):
    new_tour = tour[:]
    L = levy_step_length()
    n = len(new_tour)
    for _ in range(L):
        i, j = random.sample(range(n), 2)
        new_tour[i], new_tour[j] = new_tour[j], new_tour[i]
    return new_tour

def random_permutation(n):
    perm = list(range(n))
    random.shuffle(perm)
    return perm

def cuckoo_search_tsp(cities, n_nests=15, pa=0.25, max_iter=500, verbose=True):
    n_cities = len(cities)
    nests = [random_permutation(n_cities) for _ in range(n_nests)]
    fitness = [tour_length(tour, cities) for tour in nests]
    best_index = min(range(n_nests), key=lambda i: fitness[i])
    best_tour = nests[best_index][:]
    best_distance = fitness[best_index]
    for t in range(max_iter):
        j = random.randrange(n_nests)
        cuckoo = discrete_levy_flight(nests[j])
        cuckoo_fit = tour_length(cuckoo, cities)
        k = random.randrange(n_nests)
```

```
        if cuckoo_fit < fitness[k]:
            nests[k] = cuckoo
            fitness[k] = cuckoo_fit
        for i in range(n_nests):
            if random.random() < pa:
                nests[i] = random_permutation(n_cities)
                fitness[i] = tour_length(nests[i], cities)
        best_index = min(range(n_nests), key=lambda i: fitness[i])
        if fitness[best_index] < best_distance:
            best_tour = nests[best_index][:]
            best_distance = fitness[best_index]
        if verbose and (t % (max_iter // 10 + 1) == 0):
            print(f"Iteration {t}: Best distance so far = {best_distance:.3f}")
    return best_tour, best_distance

if __name__ == "__main__":
    print("=== Cuckoo Search Algorithm for TSP ===")
    n_cities = int(input("Enter number of cities: "))
    cities = []
    for i in range(n_cities):
        x = float(input(f"Enter x-coordinate of city {i}: "))
        y = float(input(f"Enter y-coordinate of city {i}: "))
        cities.append((x, y))
    n_nests = int(input("Enter number of nests (population size): "))
    pa = float(input("Enter discovery probability (0.0-1.0): "))
    max_iter = int(input("Enter maximum number of iterations: "))
    print("\nRunning Cuckoo Search...")
    best_tour, best_dist = cuckoo_search_tsp(
        cities, n_nests=n_nests, pa=pa, max_iter=max_iter, verbose=True
    )
    print("\n=== Result ===")
    print("Best tour (city indices):", best_tour)
    print(f"Best distance: {best_dist:.3f}")
```

Program 6

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:



Grey Wolf Optimizer

Application : Training of LLM

Pseudocode :

```
initialize_model (model_type = "GPT" , model_path = "path/to/
                                        pretrained/model")

function preprocess_input (input_text, task_type):
    if task_type == "text_generation":
        return clean_text(input_text)
    else if task_type == "question_answering":
        return process_qa_input (input_text)
    else:
        return input_text

function run_inference (processed_input, task_type):
    if task_type == "text_generation":
        output = generate_text(processed_text)
    else if task_type == "question_answering":
```

```
        return format_answer(output)
    else:
        return output

function apply_llm(input_text, task_type):
        processed_input = processes_input(input_text, task_type)
        model_output = run_inference(processed_input,
                                     task_type)
        final_output = postprocess_output(model_output,
                                          task_type)
        return final_output
```

(Distance: ~~X_α~~ = $D_\alpha = |C_1 X_\alpha - X(t)|$)

(Position update: $X(t+1) = X_\alpha - A_1 \cdot D_\alpha$)

Output:

Iteration1: Best fitness = 2.10976767

Iteration2: Best fitness = 4.338688

Iteration3: Best fitness = 3.8356 2143

Iteration4: Best fitness = 14.93900030

Iteration5: Best Fitness = 15.244925167

Best Solution: 2.17354375

Best fitness: 15.2449 2516

MG
29/9/25.

Code:

```
import random
import math

def kapur_entropy(thresholds, image):
    thresholds = sorted([int(round(t)) for t in thresholds])
    thresholds = [0] + thresholds + [256]
    hist = [0]*256
    total_pixels = 0
    for row in image:
        for pixel in row:
            hist[pixel] += 1
            total_pixels += 1
    prob = [h/total_pixels for h in hist]
    total_entropy = 0
    for i in range(len(thresholds)-1):
        start = thresholds[i]
        end = thresholds[i+1]
        P = [p for p in prob[start:end] if p>0]
        total_entropy += -sum([p*math.log(p) for p in P])
    return -total_entropy

def GWO_image(image, D, N=10, MaxIter=50, lb=0, ub=255):
    wolves = [[random.uniform(lb, ub) for _ in range(D)] for _ in range(N)]
    alpha_pos = [0]*D
    beta_pos = [0]*D
    delta_pos = [0]*D
    alpha_score = float("inf")
    beta_score = float("inf")
    delta_score = float("inf")
    for t in range(MaxIter):
        a = 2 - 2*t/MaxIter
        for i in range(N):
            fitness = kapur_entropy(wolves[i], image)
            if fitness < alpha_score:
                delta_score, delta_pos = beta_score, beta_pos[:]
                beta_score, beta_pos = alpha_score, alpha_pos[:]
                alpha_score, alpha_pos = fitness, wolves[i][:]
            elif fitness < beta_score:
                delta_score, delta_pos = beta_score, beta_pos[:]
                beta_score, beta_pos = fitness, wolves[i][:]
            elif fitness < delta_score:
                delta_score, delta_pos = fitness, wolves[i][:]
        for i in range(N):
            for d in range(D):
                r1, r2 = random.random(), random.random()
```

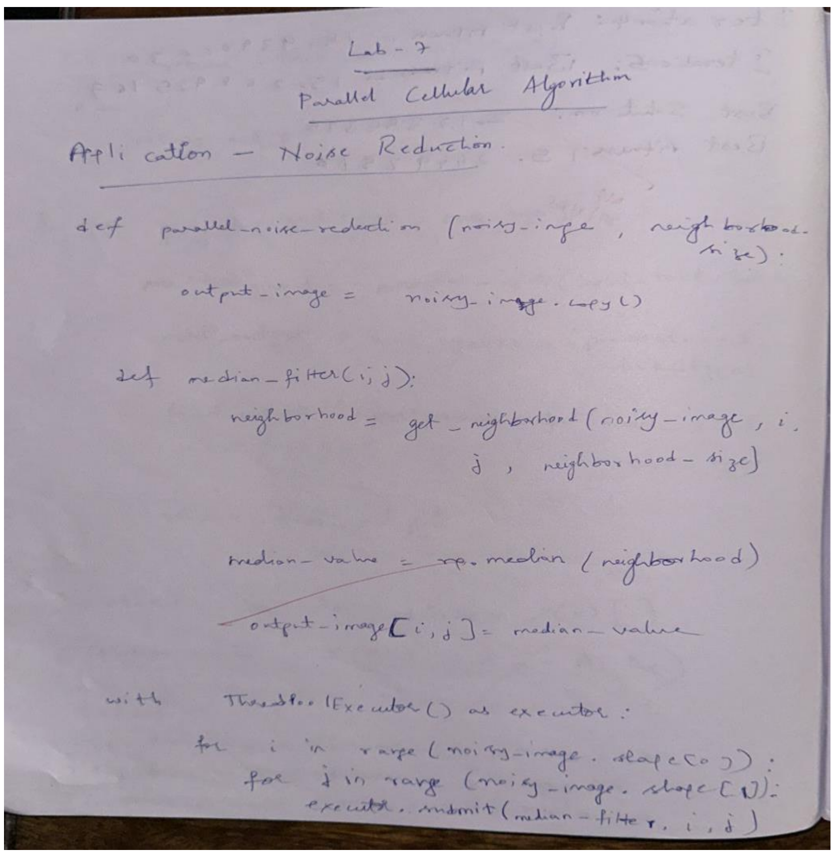
```
            A1 = 2*a*r1 - a; C1 = 2*r2
            r1, r2 = random.random(), random.random()
            A2 = 2*a*r1 - a; C2 = 2*r2
            r1, r2 = random.random(), random.random()
            A3 = 2*a*r1 - a; C3 = 2*r2
            D_alpha = abs(C1*alpha_pos[d] - wolves[i][d])
            D_beta = abs(C2*beta_pos[d] - wolves[i][d])
            D_delta = abs(C3*delta_pos[d] - wolves[i][d])
            X1 = alpha_pos[d] - A1*D_alpha
            X2 = beta_pos[d] - A2*D_beta
            X3 = delta_pos[d] - A3*D_delta
            wolves[i][d] = (X1 + X2 + X3)/3
            if wolves[i][d] < lb: wolves[i][d] = lb
            if wolves[i][d] > ub: wolves[i][d] = ub
    return [int(round(x)) for x in alpha_pos
def main():
    filename = input("Enter PGM image filename (grayscale): ")
    image = []
    with open(filename, 'r') as f:
        lines = f.readlines()
    lines = [l for l in lines if not l.startswith('#')]
    if lines[0].strip() != 'P2':
        print("Only ASCII PGM (P2) supported.")
        return
    idx = 2
    while len(image) < int(lines[1].split()[1]):
        row = list(map(int, lines[idx].split()))
        image.append(row)
        idx += 1
    D = int(input("Enter number of thresholds: "))
    N = int(input("Enter number of wolves: "))
    MaxIter = int(input("Enter maximum iterations: "))
    best_thresholds = GWO_image(image, D, N, MaxIter)
    print("Best thresholds found:", best_thresholds)
    thresholds = sorted(best_thresholds)
    thresholds = [0] + thresholds + [256]
    segmented = [[0 for _ in row] for row in image]
    for i in range(len(thresholds)-1):
        for r in range(len(image)):
            for c in range(len(image[0])):
                if thresholds[i] <= image[r][c] < thresholds[i+1]:
                    segmented[r][c] = int((i+1)*(255/(len(thresholds)-1)))
    out_file = "segmented.pgm"
    with open(out_file, 'w') as f:
        f.write("P2\n")
        f.write(f"{len(segmented[0])} {len(segmented)}\n")
        f.write("255\n")
```

```
        for row in segmented:
            f.write(' '.join(map(str,row)) + '\n')
    print(f"Segmented image saved as {out_file}")

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

Program 7

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:



Lab - 7

Parallel Cellular Algorithm

Application - Noise Reduction

```
def parallel_noise_reduction (noisy_image , neighborhood_size):
    output_image = noisy_image.copy()

    def median_filter(i, j):
        neighborhood = get_neighborhood (noisy_image, i, j, neighborhood_size)

        median_value = np.median (neighborhood)

        output_image[i, j] = median_value

    with ThreadPoolExecutor() as executor:
        for i in range (noisy_image.shape[0]):
            for j in range (noisy_image.shape[1]):
                executor.submit(median_filter, i, j)
```

```
    return output_image

def get_neighborhood(image, i, j, neighborhood_size):
    neighborhood = []

    half_size = neighborhood // 2
    for di in range(-half_size, half_size + 1):
        for dj in range(-half_size, half_size+1):
            ni, nj = i+di, j+dj

            if 0 <= ni < image.shape[0] and
               0 <= nj < image.shape[1]:
                neighborhood.append(image[ni, nj])

    return neighborhood

denoised_image = parallel_noise_reduction(noisy_image,
                          neighborhood_size = 3)
```

Output:

Original grid

```
##...
##...
.....
.....
....#
```

~~Original~~

Noisy:

```
##.##.
##..#
.....
..##.
...##
```

MG
13/10/25.

Denoised (Parallel Cellular Automation)

```
.....
.....
.....
.....
.....
```

Code:

```
import random

EMPTY = " "
TREE = "T"
BURNING = "F"

def get_neighbors(grid, i, j, neighborhood_type=8):
    rows = len(grid)
    cols = len(grid[0])
    neighbors = []
    if neighborhood_type == 4:
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    else:
        directions = [
            (-1, -1), (-1, 0), (-1, 1),
            (0, -1),          (0, 1),
            (1, -1), (1, 0), (1, 1)
        ]
    for dx, dy in directions:
        x, y = i + dx, j + dy
        if 0 <= x < rows and 0 <= y < cols:
            neighbors.append((x, y))
    return neighbors

def ForestFireModel(grid, num_iterations, prob_lightning, prob_tree_growth):
    for _ in range(num_iterations):
        new_grid = [row[:] for row in grid]
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                state = grid[i][j]
                neighbors = get_neighbors(grid, i, j, 8)
                if state == BURNING:
                    new_grid[i][j] = EMPTY
                elif state == TREE:
                    if any(grid[x][y] == BURNING for x, y in neighbors):
                        new_grid[i][j] = BURNING
                    elif random.random() < prob_lightning:
                        new_grid[i][j] = BURNING
                elif state == EMPTY:
                    if random.random() < prob_tree_growth:
                        new_grid[i][j] = TREE
        grid = new_grid
        print_grid(grid)
    return grid
```

```
def print_grid(grid):
    for row in grid:
        print(" ".join(row))
    print("-" * (2 * len(grid[0]) - 1))

if __name__ == "__main__":
    rows = int(input("Enter number of rows: "))
    cols = int(input("Enter number of columns: "))
    num_iterations = int(input("Enter number of iterations: "))
    prob_lightning = float(input("Enter probability of lightning (0-1): "))
    prob_tree_growth = float(input("Enter probability of tree growth (0-1): "))
    grid = []
    for i in range(rows):
        row = []
        for j in range(cols):
            r = random.random()
            if r < 0.6:
                row.append(TREE)
            elif r < 0.8:
                row.append(EMPTY)
            else:
                row.append(BURNING)
        grid.append(row)
    print("\nInitial Forest:")
    print_grid(grid)
    print("Simulating fire spread...\n")
    final_grid = ForestFireModel(grid, num_iterations, prob_lightning, prob_tree_growth)
    print("Final Forest State:")
    print_grid(final_grid)
```