VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Dhanush K
1BM24CS404

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 **Dhanush K (1BM24CS404),** 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.

Sowmya T Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/dhanushk240206/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 01

genetic Algorithm.

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

Ex: (1) $x \rightarrow 0-31$

$$\text{Prob} = \frac{f(x)}{\Sigma f(x)} \cdot \frac{144}{1155} = 0.1$$

$$\text{Expected output Avg} = \frac{f(x_i)}{(\Sigma f(x))} = \frac{144}{288.75} = 0.49$$

(2)

String No	Initial population	X value	fitness $f(x)=x^2$	prob	% prob	Expected output	Actual count
1	01100	12	144	0.1247	12.47	0.4987	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	181	0.3126	31.26	1.25	1
Sum			1155	1.0	100	4	
Average			288.75	0.25	25	1	
Maximum			625	0.5411	54.11	2.16	

Page 02

3. Selecting Mating Pool

String No	Mating Pool	Crossover point	Offspring after Crossover	X Value	fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289
Sum					1763
Avg					440.75
Max					729

Crossover → Crossover point is chosen randomly.

4. Mutation.

random

String No	Offspring after Crossover	Mutation after 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	00000	11011	27	729
4	10001	00101	10100	20	400
Sum					2546
Avg					636.5
Max					841

Page 03

Algorithm

1. Genetic Algorithm for optimization Problem.

→ A Search and optimization technique inspired by natural Selection, where populations evolve toward better Solutions over generations.

Applications.

→ Function optimization
→ Machine learning & Neural Networks
→ Scheduling problems
✓→ Traveling Salesman
→ Robotics & path planning.

Pseudocode for Genetic Algorithm.

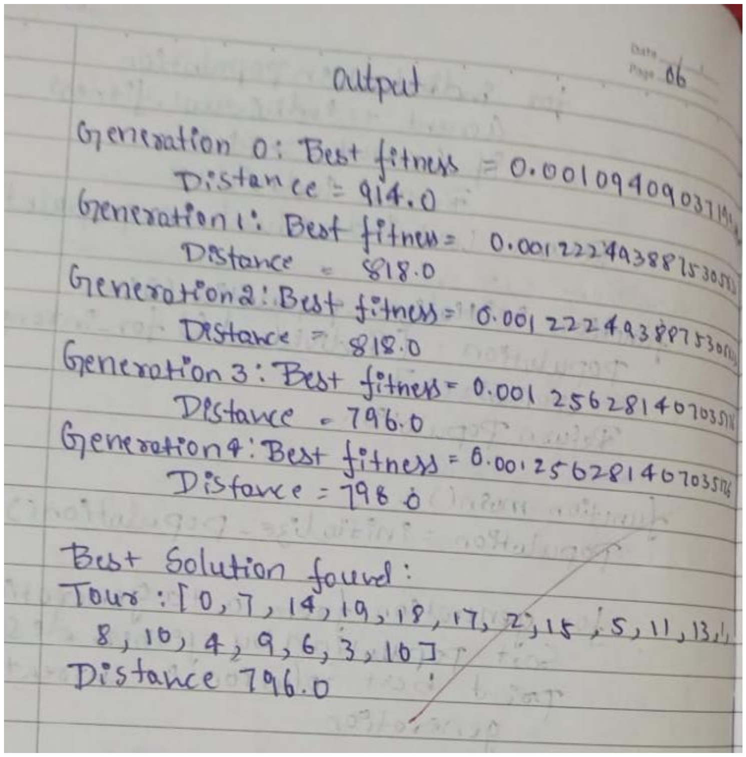
```
Define Constants:
    Pop_Size = 6
    Genome_length = 5
    generations = 5
    Mutation_Rate = 10%

Class Individual:
    function __init__():
        genome = random list of 0s & 1s with
        length Genome_Length
        fitness = calculate_fitness()

    function calculate_fitness():
        total_distance = 0
        for i from 0 to num_cities - 1:
            total_distance += Distance_
```

Page 04

```
            matrix[genome[i]][genome[i+1]]
        # Add distance from last city to the
        first city (return to start)
        total distance += Distance_Matrix
            [genome[NUM_Cities - 1]][genome[0]]
        Return 1 / total_distance # Fitness
            is the inverse of distance

function mutate():
    If random() < Mutation_Rate:
        Swap i and j in genome
        fitness = calculate_fitness()

function Crossover (parent1, parent2):
    Child1 = Crossover_genes (p1, p2)
    Child2 = Crossover_genes (p2, p1)
    Child1.fitness = Child1.calculate_fitness()
    Child2.fitness = Child2.calculate_fitness()
    Return Child1, child2

function Crossover_genes(p1, p2)
    Return a new individual (child) from
    parent1 & Parent2.

Function Selection(populations):
    total_fitness = Sum of all fitness
        values in population
    pick = random integer b/w 0 &
            total fitness
    current = 0
```

Page 05

```
    for individual in population
        current += individual.fitness
        if current > pick:
            Return individual
    Return last individual in population

function initialize_population():
    Population = [Individual() for _ in range
        (pop_size)]
    Return Population

function main():
    Population = initialize_population()

    for generation from 0 to generation-1:
        Sort population by fitness descending
        print best Solution in current
            generation

        new_population = [best individual(s)
            from current population]

        while new_population is less than pop_size
            p1 = Selection(population)
            p2 = Selection(population)
            child1, child2 = parent1.crossover(p2
            child1.mutate()
            Child.mutate()
            Add child1, child2 to new_populat

        population = new_population

call main()
```


Output
Page 06
Generation 0: Best fitness = 0.00109409037
Distance = 914.0
Generation 1: Best fitness = 0.0012224938875305
Distance = 818.0
Generation 2: Best fitness = 0.0012224938875305
Distance = 818.0
Generation 3: Best fitness = 0.00125628140703517
Distance = 796.0
Generation 4: Best fitness = 0.00125628140703517
Distance = 798.0

Best Solution found:
Tour: [0, 7, 14, 19, 18, 17, 2, 15, 5, 11, 13,
8, 10, 4, 9, 6, 3, 10]
Distance 796.0

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()
```

Program 2

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:

Page 07

Lab-01

Optimization via Gene Expression Algorithm.

→ GEA are inspired by the biological process of gene expression, where genetic information is translated into function solutions.

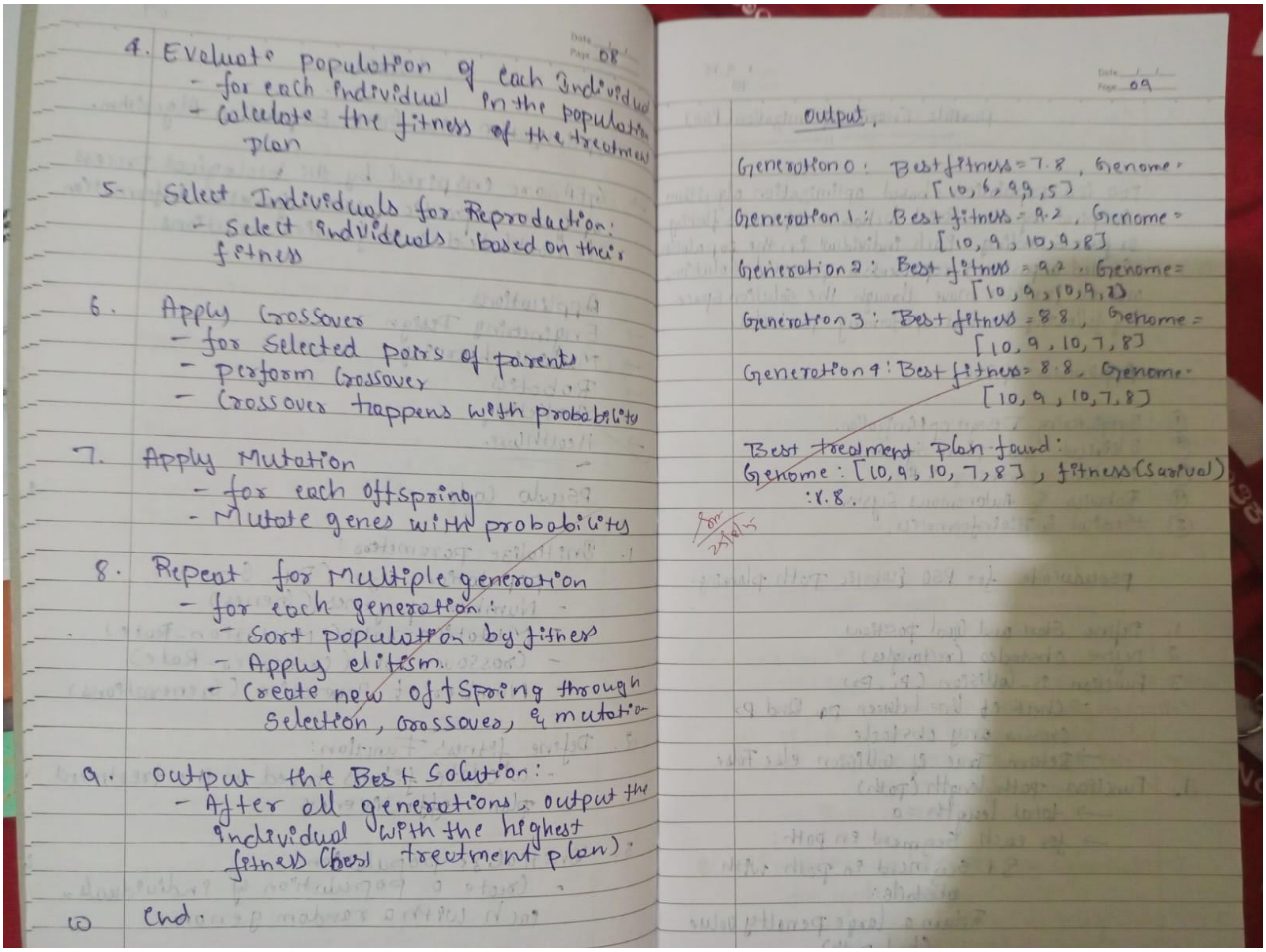
Applications.
→ Engineering Design
→ Machine Learning.
→ Robotics
→ Marketing.
→ Healthcare.

Pseudo Code:

1. Initialize Parameters:
   - population size (POP_SIZE)
   - Number of genes (Genes)
   - Mutation rate (Mutation_Rate)
   - Crossover rate (Crossover_Rate)
   - Number of generation (Generations)

2. Define fitness Function:
   - calculate fitness based on the treatment plan's effectiveness

3. Initialize population
   - Create a population of individuals, each with a random genome


Page 08

4. Evaluate Population of each Individual
- for each individual in the population
- Calculate the fitness of the treatment plan

5. Select Individuals for Reproduction:
- Select individuals based on their fitness

6. Apply Crossover
- for selected pairs of parents:
- perform crossover
- Crossover happens with probability

7. Apply Mutation
- for each offspring
- Mutate genes with probability

8. Repeat for multiple generation
- for each generation:
  - Sort population by fitness
  - Apply elitism.
  - Create new offspring through selection, crossover, & mutation

9. Output the Best Solution:
- After all generations, output the individual with the highest fitness (best treatment plan).

10 End.

Page 09

Output.

Generation 0: Best fitness = 7.8, Genome = [10, 6, 9, 9, 5]
Generation 1: Best fitness = 9.2, Genome = [10, 9, 10, 9, 8]
Generation 2: Best fitness = 9.2, Genome = [10, 9, 10, 9, 8]
Generation 3: Best fitness = 8.8, Genome = [10, 9, 10, 7, 8]
Generation 4: Best fitness = 8.8, Genome = [10, 9, 10, 7, 8]

Best treatment plan found:
Genome: [10, 9, 10, 7, 8], fitness (Survival): 8.8.

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):
```

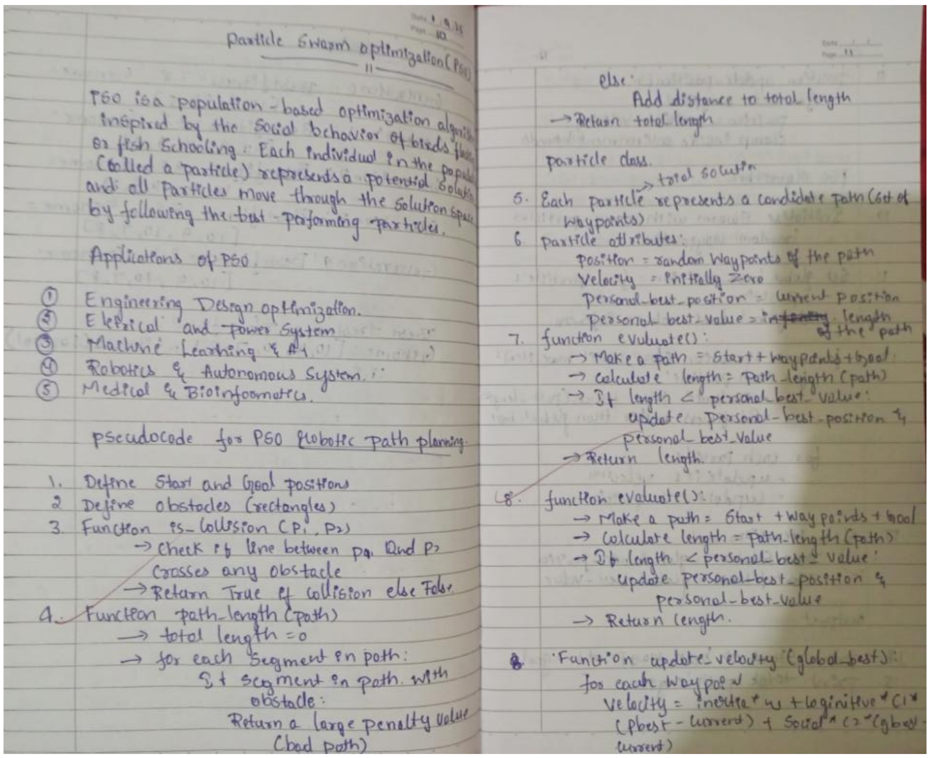
```
        return fitness_function(self.genome)

    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:



Particle Swarm optimization (PSO)

PSO is a population-based optimization algorithm inspired by the social behavior of birds flocking or fish schooling. Each individual in the population (called a particle) represents a potential solution and all particles move through the solution space by following the best-performing particles.

Applications of PSO.

1. Engineering Design optimization.
2. Electrical and Power System
3. Machine Learning & AI
4. Robotics & Autonomous System.
5. Medical & Bioinformatics.

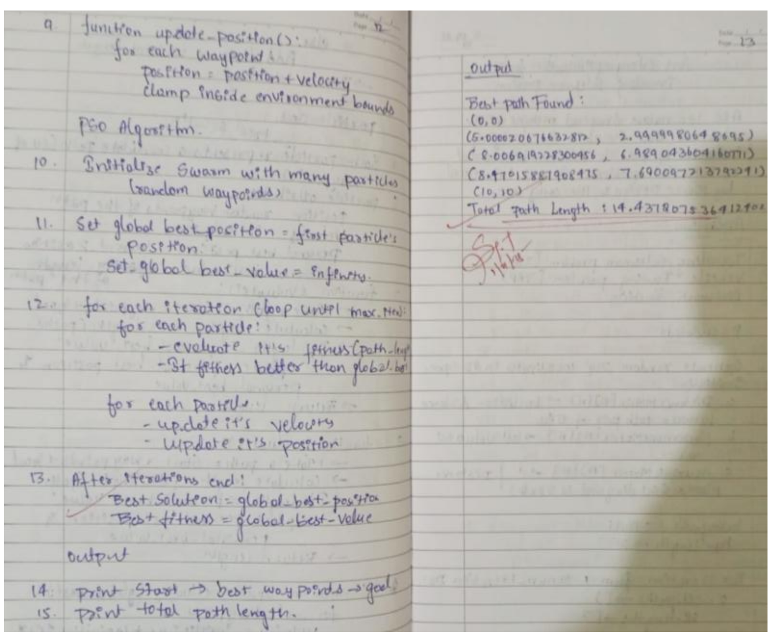
pseudocode for PSO Robotic path planning.

1. Define Start and Goal positions
2. Define obstacles (rectangles)
3. Function is_collision (P1, P2)
   → check if line between P1 and P2 crosses any obstacle
   → Return True if collision else False
4. Function path_length (path)
   → total length = 0
   → for each segment in path:
      If segment in path with obstacle:
         Return a large penalty value (bad path)
      else:
         Add distance to total length
   → Return total length

particle class.

5. Each particle represents a candidate path (set of waypoints) → total solution
6. particle attributes:
   position = random waypoints of the path
   velocity = initially zero
   personal_best_position = current position
   personal_best_value = ~~infinity~~ length of the path
7. function evaluate():
   → Make a path = Start + waypoints + Goal
   → calculate length = path_length (path)
   → If length < personal_best_value:
      update personal_best_position & personal_best_value
   → Return length.
8. function evaluate():
   → Make a path = Start + waypoints + Goal
   → Calculate length = path_length (path)
   → If length < personal_best_value:
      update personal_best_position & personal_best_value
   → Return length.

8. Function update_velocity (global_best):
   for each waypoint
      velocity = inertia * w + cognitive * c1 * (pbest - current) + social * c2 * (gbest - current)



9. function update_position():
   for each waypoint:
      position = position + velocity
      clamp inside environment bounds

PSO Algorithm.

10. Initialize Swarm with many particles (random waypoints)
11. Set global best_position = first particle's position
    Set global_best_value = infinity.
12. for each iteration (loop until max_iter):
    for each particle:
       - evaluate it's fitness (path_length)
       - If fitness better than global_best
    for each particle
       - update it's velocity
       - update it's position
13. After iterations end:
    Best Solution = global_best_position
    Best fitness = global_best_value

output

14. Print Start → best waypoints → goal
15. Print total path length.

output

Best path Found:
(0, 0)
(5.00002067663281?, 2.999998064 8695)
(8.006919778300956, 6.989043604160771)
(8.470158871908475, 7.690097713792791)
(10, 10)
Total path Length : 14.4378075364127??

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:

Date 8/09/25
Page 14

Ant Colony optimization for the Travelling Salesman problem.

ACO is a nature-inspired metaheuristic algorithm that simulates the foraging behavior of ants to solve combinatorial optimization problems. It was introduced by Marco Dorigo in the early 1990's.

Applications

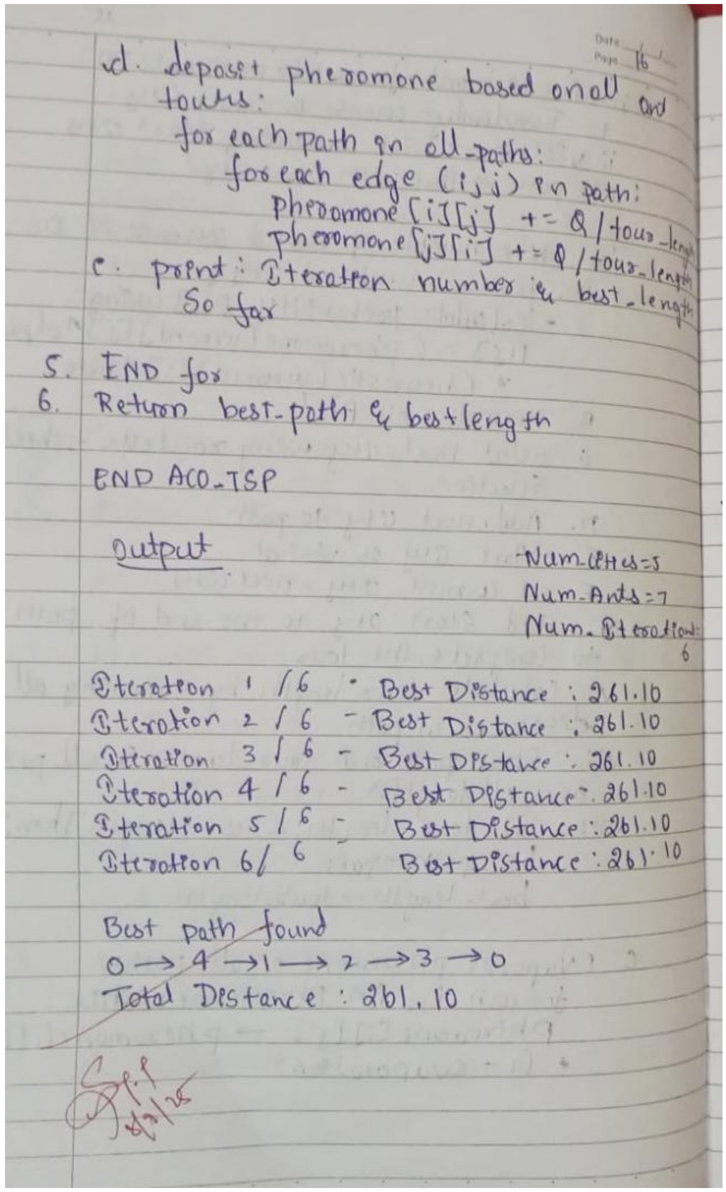
1. Traveling Salesman problem [TSP]
2. Vehicle Routing problem [VRP]
3. Network Routing

Pseudocode

1. Generate random city coordinates in 2D space
2. Initialize:
   a. DistanceMatrix[N][N] → Euclidean distance between each pair of cities.
   b. PheromoneMatrix[N][N] → all values set to 1
   c. HeuristicMatrix[N][N] → 1 / Distance Matrix (set diagonal to 0 or ∞)
3. best_path ← NULL
   best_length → ∞
4. For iteration from 1 to num_iteration DO:
   a. all_paths → []
      all_lengths → []

Date __/__/__
Page 15

   b. For each ant from 1 to num_ants DO:
      i. Randomly choose a start city
      ii. visited → set containing start city
      iii. path → list with start city.
      iv. While number of visited cities < N DO:
         A. for each unvisited city j:
            - Calculate probability P[j] using:
              P[j] = (pheromone[current][j]^alpha) * (heuristic[current][j]^beta)
         B. Normalize all P[j]
         C. Select next city using roulette-wheel selection
         D. Add next city to path
         E. Mark city as visited
         F. Set current city = next city
      v. Add start city to the end of path to complete the tour
      vi. Calculate tour_length by summing all distances in path
      vii. Store path and tour_length in all_paths and all_lengths
      viii. If tour length < best length Then:
            best_path → path
            best_length → tour length
   c. Evaporate pheromone on all paths:
      for each i, j in pheromone matrix:
      pheromone[i][j] → pheromone[i][j] * (1 - evaporation)


d. deposit pheromone based on all ant tours:
for each path in all_paths:
for each edge (i, j) in path:
pheromone[i][j] += Q / tour_length
pheromone[j][i] += Q / tour_length
e. print: Iteration number & best_length so far

5. END for
6. Return best_path & best length

END ACO_TSP

Output

Num_cities = 5
Num_Ants = 7
Num_Iterations: 6

Iteration 1 / 6 - Best Distance : 261.10
Iteration 2 / 6 - Best Distance : 261.10
Iteration 3 / 6 - Best Distance : 261.10
Iteration 4 / 6 - Best Distance : 261.10
Iteration 5 / 6 - Best Distance : 261.10
Iteration 6 / 6 - Best Distance : 261.10

Best path found
0 → 4 → 1 → 2 → 3 → 0
Total Distance : 261.10

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)
```

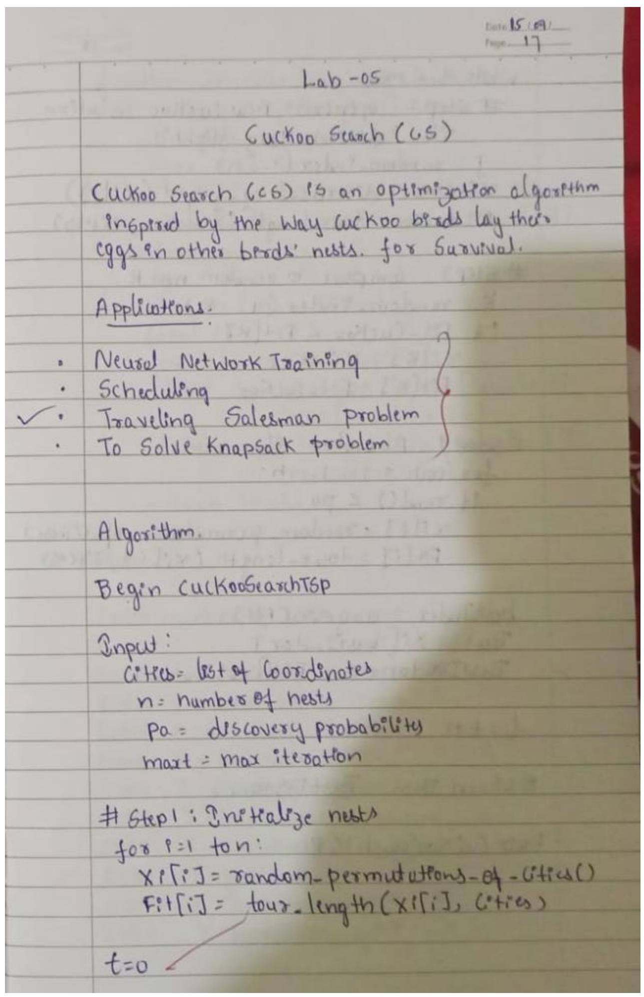
```
    dist = np.zeros((n_cities, n_cities))
    for i in range(n_cities):
        for j in range(n_cities):
            dist[i][j] = calculate_distance(cities[i], cities[j])
    pheromone = np.ones((n_cities, n_cities)) * 0.1
    best_path = None
    best_path_length = float('inf')
    for _ in range(n_iterations):
        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)
```

Program 5

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:



Date 15/9/
Page 17

Lab -05

Cuckoo Search (CS)

Cuckoo Search (CS) is an optimization algorithm inspired by the way cuckoo birds lay their eggs in other birds' nests. for Survival.

Applications.

- Neural Network Training
- Scheduling
- Traveling Salesman problem
- To Solve Knapsack problem

Algorithm.

Begin CuckooSearchTSP

Input:
Cities = list of Coordinates
n = number of nests
pa = discovery probability
maxt = max iteration

Step1 : Initialize nests
for i=1 to n:
X[i] = random_permutations_of_cities()
Fit[i] = tour_length(X[i], Cities)

t=0

Page 18

```
while t < Maxt:
  # Step 2: Generate new cuckoo solution
      via "discrete levy flight"
   j = random_index(1..n)
   Xcuckoo = discrete_levy_flight(Xi[j])
   Fit_cuckoo = tour_length(Xcuckoo, cities)

 # Step 3: Compare to random nest K
   K = random_index(1...n)
   if Fit_Cuckoo < Fit[K]:
      Xi[K] = Xcuckoo
      Fit[K] = Fit_Cuckoo

 # Step 4: Abandon some nests
   for each i in 1...n:
     if rand() < pa:
        Xi[i] = random_permutation_of_cities()
        Fit[i] = tour_length(Xi[i], cities)

 bestIndex = argmin(Fit)
 Best = Xi[bestIndex]
 BestDistance = Fit[bestIndex]

 t = t+1

Return Best, BestDistance

END CuckooSearchTSP
```

Page 19

```
function discrete_levy_flight(tour):
    L = levy_step_length()
    new_tour = copy(tour)
    for s=1 to L:
       i, j = random distinct position
       swap(new_tour[i], new_tour[j])
    return new_tour

tour_length(tour, cities):
    total = 0
    for k=1 to len(tour)-1
     total += distance(cities[tour[k]],
                  cities[tour[k+1]])
    total += distance(cities[tour[-1]],
                  cities[tour[0]])
    return total.
```

Output

```
Enter number of cities : 5

Enter X-coordinates of city 0 : 0
Enter Y-coordinates of city 0 : 0
Enter X-coordinates of city 1 : 1
Enter Y-coordinates of city 1 : 5
Enter X-coordinates of city 2 : 5
Enter Y-coordinates of city 2 : 2
Enter X-coordinates of city 3 : 6
Enter Y-coordinates of city 3 : 6
Enter X-coordinates of city 4 : 8
Enter Y-coordinates of city 4   3
```

Page 20

```
Enter number of nests : 10
Enter discovery probability : 0.25
Enter maximum number of iteration : 100

Running Cuckoo Search
Iteration 0 : 27.311
         11 : 24.583
         22 : 22.437
         33 : 21.975
         44 : 21.975
         55 : 20.906
         66 : 20.906
         77 : 20.906
         88 : 20.906
         99 : 20.906

Best tour : [0, 2, 4, 3, 1]
Best distance : 20.906
```

Code:

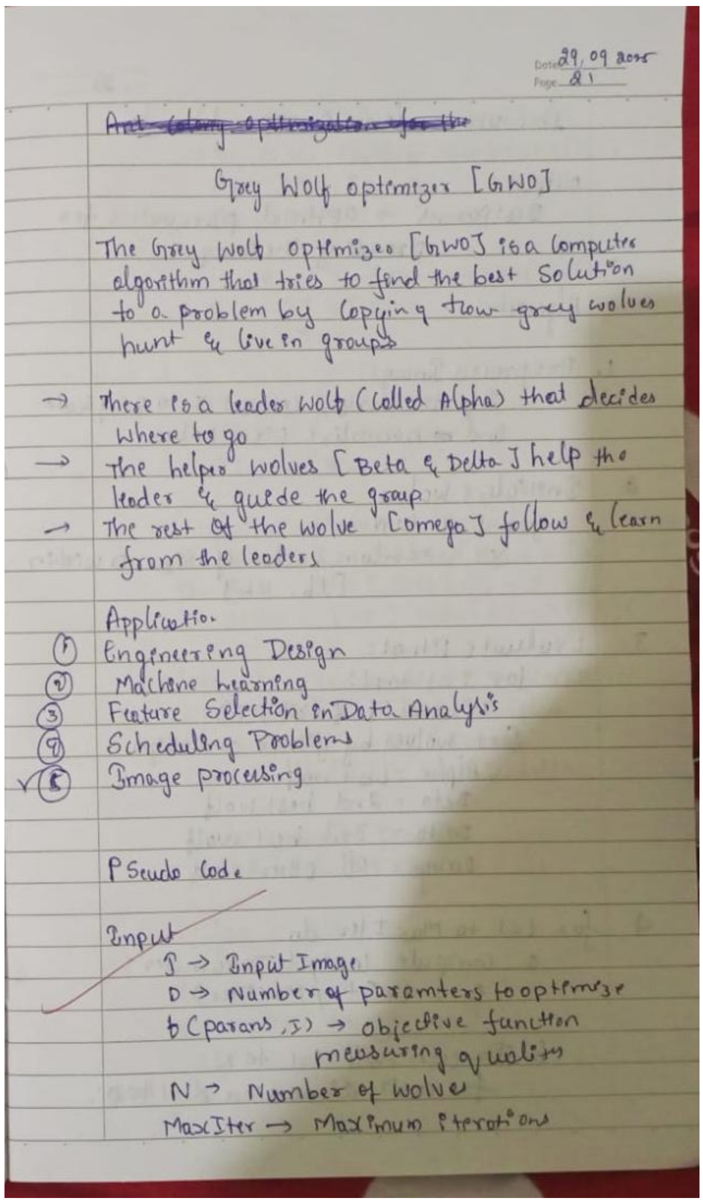
```
import random
import math

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:



Date 29, 09 2025
Page 21

~~Ant colony optimization for the~~

Grey Wolf optimizer [GWO]

The Grey Wolf optimizer [GWO] is a computer algorithm that tries to find the best solution to a problem by copying how grey wolves hunt & live in groups

→ There is a leader wolf (called Alpha) that decides where to go
→ The helper wolves [Beta & Delta] help the leader & guide the group
→ The rest of the wolve [omega] follow & learn from the leaders

Application
1. Engineering Design
2. Machine learning
3. Feature Selection in Data Analysis
4. Scheduling Problems
5. Image processing

Pseudo code

Input
I → Input Image
D → Number of paramters to optimize
f(params, I) → objective function measuring quality
N → Number of wolve
MaxIter → Maximum iterations

lb, ub → lower & upper bounds

Output:
Best Params → optimal parameters for processing the image

Algorithm:

1. Preprocess Image:
   If necessary, convert I to gray scale and normalize pixel values.

2. Initialize wolves:
   for i = 1 to N
   Xi = random vector of size D within [lb, ub]

3. Evaluate Fitness:
   for i = 1 to N:
   Fitness[i] = f(Xi, I)
   Sort wolves by fitness and assign:
   Alpha = best wolf
   Beta = 2nd best wolf
   Delta = 3rd best wolf
   Omega = all other wolf

4. for t = 1 to MaxIter do
   a. compute control parameter:
   a = 2 - (2 * t / MaxIter)

   b. For each wolf i = 1 to N:
   for each dimension d = 1 to D:

Generate random co-efficients
r1, r2 = random numbers in [0, 1]
A1 = 2 * a * r1 - a
C1 = 2 * r2

r1, r2 = random numbers in [0, 1]
A2 = 2 * a * r1 - a
C2 = 2 * r2

r1, r2 = random nubers in [0, 1]
A3 = 2 * a * r1 - a
C3 = 2 * r2

Computer distances to leaders
D_alpha = |C1 * Alpha[d] - Xi[d]|
D_beta = |C2 * Beta[d] - Xi[d]|
D_delta = |C3 * Delta[d] - Xi[d]|

Compute new candidate position
X1 = Alpha[d] - A1 * D_alpha
X2 = Beta[d] - A2 * D_beta
X3 = Delta[d] - A3 * D_delta

update wolf position
Xi[d] = (X1 + X2 + X3) / 3

End for

clip to bounds
Xi = clip(Xi, lb, ub)

End for

c. Re-evaluate fitness
   for i = 1 to N:
   fitness[i] = f(Xi, I)

d. update Alpha, Beta, Delta by Sorting wolves again

5. Return Alpha as BestParams.
6. Apply BestParams to image I to obtain processed image

Output
Enter PGM image filename: input.pgm
Enter number of thresholds: 2
Enter number of wolves: 10
Enter maximum iteration: 50

Best Solution [85, 170]
The images is saved on Segmented.pgm.

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()
```

<u>Program 7</u>

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 13, 10, 25
Page 25

Lab-07

parallel cellular Algorithm

→ Imagine a big group of tiny cells arranged in a grid. Each cell is like a little problem solver trying to find the best answer to a problem. These cells work at the same time (in parallel) and talk to their neighbors to improve their solution. Over time, by sharing information & updating themseleves, the whole group finds a really good solution.

Application
→ optimization problem
→ Image processing
→ pattern Recognition
→ Robotics
→ Modeling Natural Systems [Forest Fire System]
→ parallel simulation
→ Scheduling problem.

<u>pseudocode</u>

Input
Grid: 2D array representing the forest
num-iterations: Number of iterations to simulate the fire spread
prob-lightning: Probability that a tree catches fire
prob-tree-growth: probability that an empty space grows a tree

Output:
Grid: Final State of the forest after the Simulation

~~Psed~~ procedure ForestFireModel (Grid, num-iterations, prob-lightning, prob-tree-growth):

Step1 : Initialize the grids with trees. empty spaces, & burning trees.

Step2 : Start the Simulation for a given number of iteration
for iteration = 1 to num.iterations do

newGrid = copy (Grid)

Step3 : Update each cell in the grid based on it's state and neighbors
for each cell (i, j) in Grid do

Step4 : Get the State of the current cell
State = Grid [i][j]

neighbors = GetNeighbors (Grid, i, j, 8)

Step5 : update the state based on the current state and neighbors' states
if State == Burning:
NewGrid[i][j] = Empty

else if State == Tree:

if any neighbor in neighbors is Burning:
NewGrid [i][j] = Burning

else if random() < prob-lightning:
newGrid [i][j] = Burning

else if State == Empty:
if random() < prob-tree-growth:
NewGrid [i][j] = Tree

Grid = NewGrid

return Grid.

Function GetNeigh (Grid, i, j, neighborhood-type):
neighbors = []
rows = number of rows in Grid
Cols = number of cloumns in Grid

if neighborhood type == 4:
neighbors (up, down, left, right)
neighbors = [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]

else if neighborhood-typ == 8:
neighbors = [
(i-1, j-1), (i-1, j), (i-1, j+1),
(i, j-1), (i, j+1),
(i+1, j-1), (i+1, j), (i+1, j+1)
]
neighbors = FilterValidNeighbors (neighbors, Grid)

return ~~n~~neighbors

Function FilterValid (neighbors, Grid):
valid-neighbors = [ ]
rows = num of rows in Grid
cols = num of columns in Grid

for each neighbor (x, y) in neighbors do
if x >= 0 and x < rows & y >= 0 & y < cols:
Valid-neighbors.append (x, y)

return valid_neighbors

Output:

Enter number of rows : 5
Enter number of columns : 5
Enter number of iteration : 5
Enter prob of lightning (0-1) : 0.01
Enter prob of tree growth (0-1) : 0.05

Initial Forest

T	.		F	T
T	T	T	F	T
F	F			T
T	T	T	T	T
	F	T		F

Simulating fire spread.

T				F
F	F	F		F
				F
F	F	F	F	F
		F		

F	T			
			T	
			T	

	F			
			T	
			T	

			T	
			T	

Final Forest State

			T	
			T	
T				T

13/10/25

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)
```