VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Pramitha J O (1BM22CS240)

in partial fulfilment 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
Sep-2024 to Jan-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 **Pramitha J 0(1BM22CS240),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Selva Kumar Professor & HOD Department of CSE, BMSCE

Index

GitHub Link: https://github.com/pramitha-jo/BIS_1BM23CS240.git

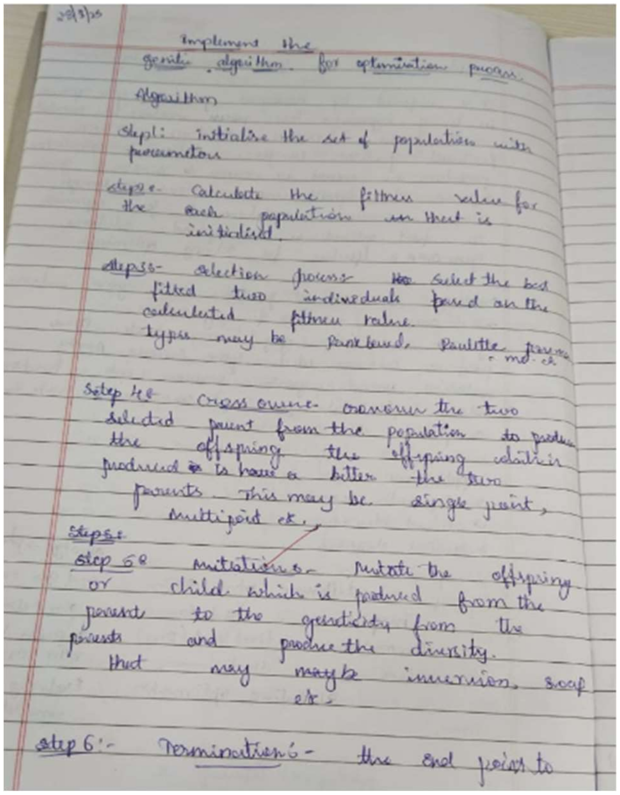
Program 1

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.

Implementation Steps:
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:



29/3/25

Implement the genetic algorithm for optimization process

Algorithm

Step1: initialise the set of population with parameters

step2:- Calculate the fitness value for the each population in that is initialised.

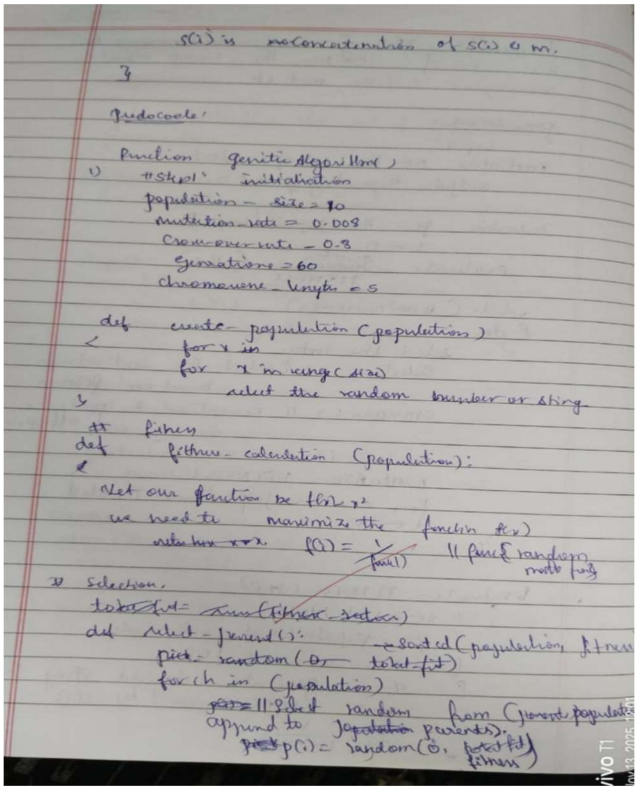
step3:- selection process- select the best fitted two individuals based on the calculated fitness value. types may be Rank based, Roulette, tournament etc.

Step 4:- Cross over- crossover the two selected parent from the population to produce the offspring the offspring which is produced is have a better the two parents. This may be single point, multipoint etc.

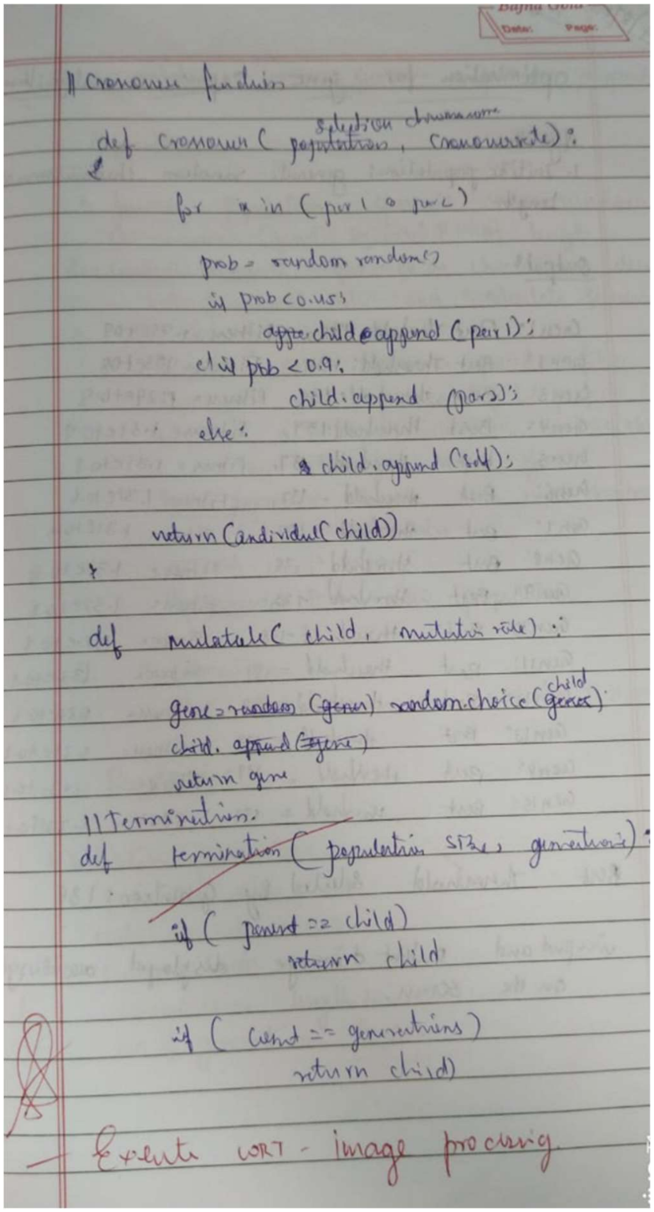
Steps:

Step 5:- Mutation:- Mutate the offspring or child which is produced from the parent to the genetically from the parents and produce the diversity. that may maybe inversion, swap etc.

step 6:- Termination:- the end point to

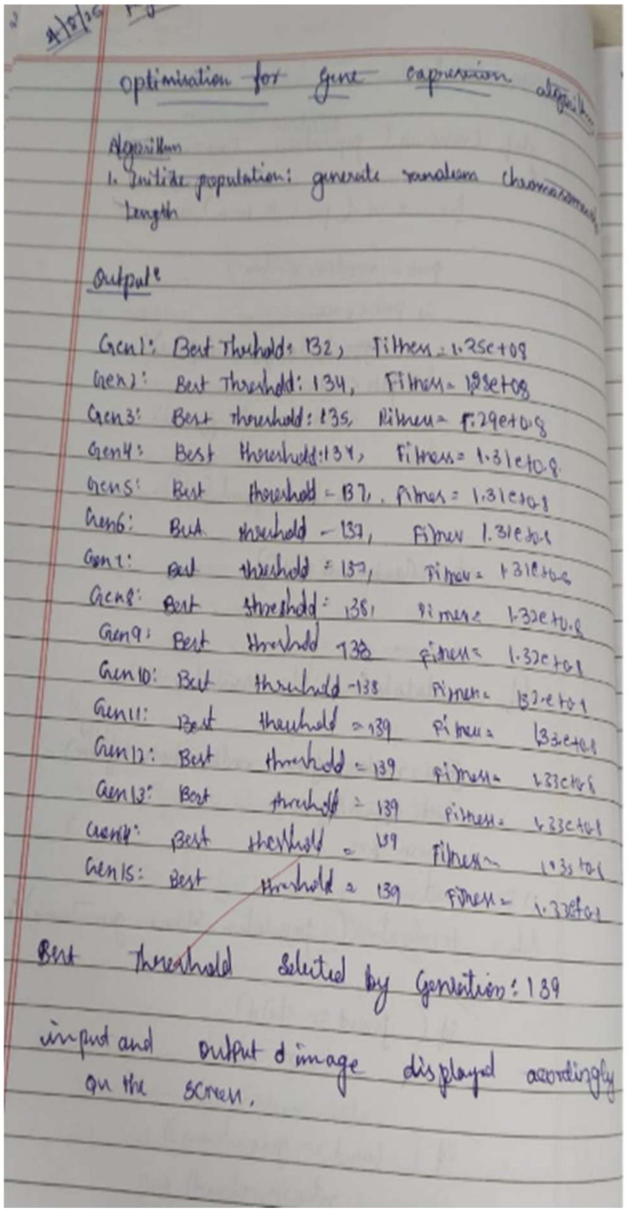


```
s(i) is no concatenation of s(i) ∪ m.
}

Pseudocode:

Function geneticAlgorithm()
1)   #step1: initialization
     population_size = 10
      mutation_rate = 0.008
      Cross-over rate = 0.8
       generations = 60
      chromosome_length = 5

 def    create_population(population)
 <         for x in
           for  x in range(size)
                select the random number or string
}
 #  fitness
def     fitness_calculation (population):
{
 Let our function be f(x)= x^2
 we need to   maximize the   function f(x)
      f(i) = 1/func(i)   // func{random math func}
2) Selection.
   total_fit = sum(fitness_values)
   def select_parent():          → sorted(population, fitness
       pick = random(0, total_fit)
       for ch in (population)
           // select random   from (parent population
       append to  parents).
           p(i) = random(0, total fitness)
```



```
// crossover function
                        selection chromosome
def crossover( population, crossoverrate):
{
      for x in ( par1 o par2)

         prob = random random()
         if prob <0.45:
               ~~appe~~ child.append (par1);
         elif prob <0.9:
               child.append (par2);
         else:
               child.append (self);

      return (individual(child))

}

def   mutate( child, mutation rate) :
                                                   child
      gene = random (genes) random.choice(genes)
      child. append(gene)
      return gene
// Termination.
def   termination( population size, generations):

      if ( parent == child)
             return child

      if ( count == generations)
             return child)
```

Execute WRT - image procesing



4/8/25

optimization for gene expression algorith...

Algorithm
1. Initialize population: generate random chromosomes
Length

Output:

```
Gen1: Best Threshold: 132,  Fitness: 1.25e+09
Gen2: Best Threshold: 134,  Fitness: 1.28e+08
Gen3: Best threshold: 135,  Fitness: 1.29e+0.8
Gen4: Best threshold: 134,  Fitness: 1.31e+0.8
Gen5: Best threshold = 137, Fitness: 1.31e+0.8
Gen6: Best threshold = 137, Fitness 1.31e+0.8
Gen7: Best threshold = 137, Fitness: 1.31e+0.8
Gen8: Best threshold: 138, Fitness: 1.32e+0.8
Gen9: Best threshold 138  Fitness: 1.32e+0.8
Gen10: Best threshold -138 Fitness: 132e+0.8
Gen11: Best threshold = 139 Fitness: 133e+0.8
Gen12: Best threshold = 139 Fitness: 1.33e+0.8
Gen13: Best threshold = 139 Fitness: 1.33e+0.8
Gen14: Best threshold = 139 Fitness: 1.33e+0.8
Gen15: Best threshold = 139 Fitness: 1.33e+0.8
```

Best Threshold Selected by Generation: 139

input and output d image displayed accordingly on the screen.

Code:

```
import cv2 import numpy as np import random import matplotlib.pyplot as plt

# Genetic Algorithm Parameters

POP_SIZE = 20          # Population size
N_GENERATIONS = 30     # Number of generations
MUTATION_RATE = 0.2    # Probability of mutation
IMG_PATH = r"C:\Users\prami\OneDrive\Documents\Family.jpg"

def fitness(image, threshold):
    """Calculate fitness based on between-class variance (Otsu’s method)."""
    _, binary = cv2.threshold(image, threshold, 255, cv2.THRESH_BINARY)


    hist = cv2.calcHist([image], [0], None, [256], [0,256]).ravel()
    total_pixels = image.size
    w0 = np.sum(hist[:threshold]) / total_pixels
    w1 = np.sum(hist[threshold:]) / total_pixels

    m0 = np.sum([i*hist[i] for i in range(threshold)]) / (np.sum(hist[:threshold])+1e-6)
    m1 = np.sum([i*hist[i] for i in range(threshold, 256)]) / (np.sum(hist[threshold:])+1e-6)
    variance = w0 * w1 * (m0 - m1) ** 2
    return variance

def initialize_population():
    return [random.randint(0, 255) for _ in range(POP_SIZE)]

def selection(population, scores):
    """Roulette wheel selection."""
    scores = np.array(scores)
    probs = scores / (scores.sum() + 1e-6)
    idx = np.random.choice(range(POP_SIZE), size=2, p=probs)
    return population[idx[0]], population[idx[1]]

def crossover(p1, p2):
    """Single point crossover."""
    point = random.randint(0, 7)
    mask = (1 << point) - 1  child1 = (p1 & mask) | (p2 & ~mask)
    child2 = (p2 & mask) | (p1 & ~mask)
    return child1, child2

def mutation(threshold):
    """Random bit flip mutation."""
    if random.random() < MUTATION_RATE:
        bit = 1 << random.randint(0, 7)
        threshold ^= bit
    return max(0, min(255, threshold))

 =================================
     Main GA Optimization
=================================
def genetic_thresholding(image):
    population = initialize_population()

    for generation in range(N_GENERATIONS):
    scores = [fitness(image, t) for t in population]

    new_population = []       for _ in range(POP_SIZE // 2):
    p1, p2 = selection(population, scores)
    c1, c2 = crossover(p1, p2)
    new_population.append(mutation(c1))
    new_population.append(mutation(c2))
```

```
        population = new_population
        best_score = max(scores)
        best_threshold = population[np.argmax(scores)]
        print(f"Gen {generation+1}:Best Threshold = {best_threshold}, Fitness = {best_score:.2f}")

        return best_threshold

if __name__ == "__main__":
        # Load image in grayscale
        img = cv2.imread(IMG_PATH, cv2.IMREAD_GRAYSCALE)

        best_t = genetic_thresholding(img)

        # Apply best threshold
        final_img = cv2.threshold(img, best_t, 255, cv2.THRESH_BINARY)

        # Show result
        plt.subplot(1,2,1), plt.imshow(img, cmap="gray"), plt.title("Original")
        plt.subplot(1,2,2), plt.imshow(final_img, cmap="gray"), plt.title(f"GA Threshold {best_t}")
        plt.show()
```

Output:

```
Gen 1: Best Threshold = 73, Fitness = 2550.92
Gen 2: Best Threshold = 6, Fitness = 2550.92
Gen 3: Best Threshold = 146, Fitness = 2543.74
Gen 4: Best Threshold = 203, Fitness = 2527.72
Gen 5: Best Threshold = 134, Fitness = 2543.74
Gen 6: Best Threshold = 70, Fitness = 2543.74
Gen 7: Best Threshold = 114, Fitness = 2549.20
Gen 8: Best Threshold = 66, Fitness = 2552.17
Gen 9: Best Threshold = 82, Fitness = 2552.17
Gen 10: Best Threshold = 82, Fitness = 2543.74
Gen 11: Best Threshold = 144, Fitness = 2543.74
Gen 12: Best Threshold = 178, Fitness = 2543.74
Gen 13: Best Threshold = 178, Fitness = 2408.85
Gen 14: Best Threshold = 210, Fitness = 2408.85
Gen 15: Best Threshold = 147, Fitness = 2537.01
Gen 16: Best Threshold = 146, Fitness = 2537.01
Gen 17: Best Threshold = 128, Fitness = 2537.01
Gen 18: Best Threshold = 148, Fitness = 2537.01
Gen 19: Best Threshold = 144, Fitness = 2537.01
Gen 20: Best Threshold = 184, Fitness = 2537.01
Gen 21: Best Threshold = 136, Fitness = 2537.01
Gen 22: Best Threshold = 144, Fitness = 2537.01
Gen 23: Best Threshold = 183, Fitness = 2537.01
Gen 24: Best Threshold = 140, Fitness = 2537.01
Gen 25: Best Threshold = 182, Fitness = 2537.01
Gen 26: Best Threshold = 170, Fitness = 2537.01
Gen 27: Best Threshold = 138, Fitness = 2532.83
Gen 28: Best Threshold = 136, Fitness = 2527.72
Gen 29: Best Threshold = 136, Fitness = 2488.61 Gen 30: Best Threshold = 154, Fitness = 2527.72
```

Program 2

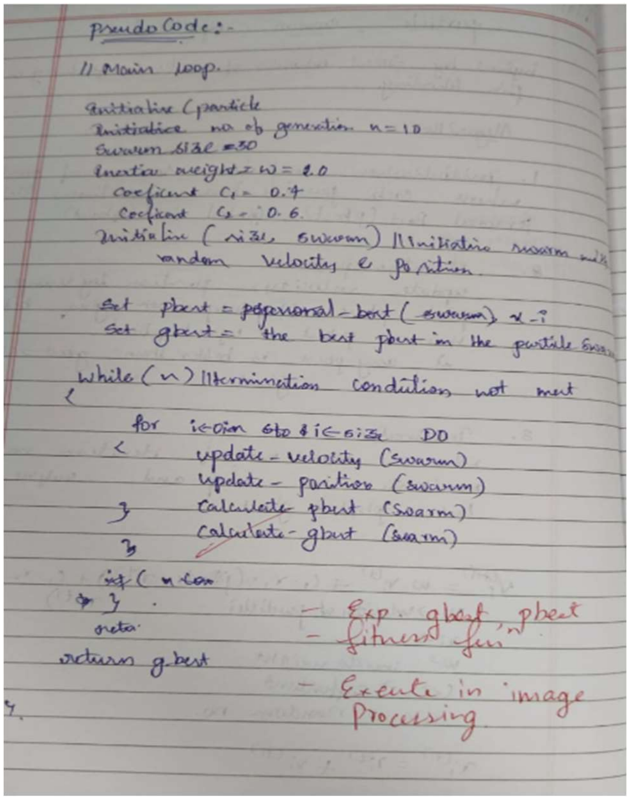
Particle Swarm Optimization for Function Optimization:

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

Implementation Steps:

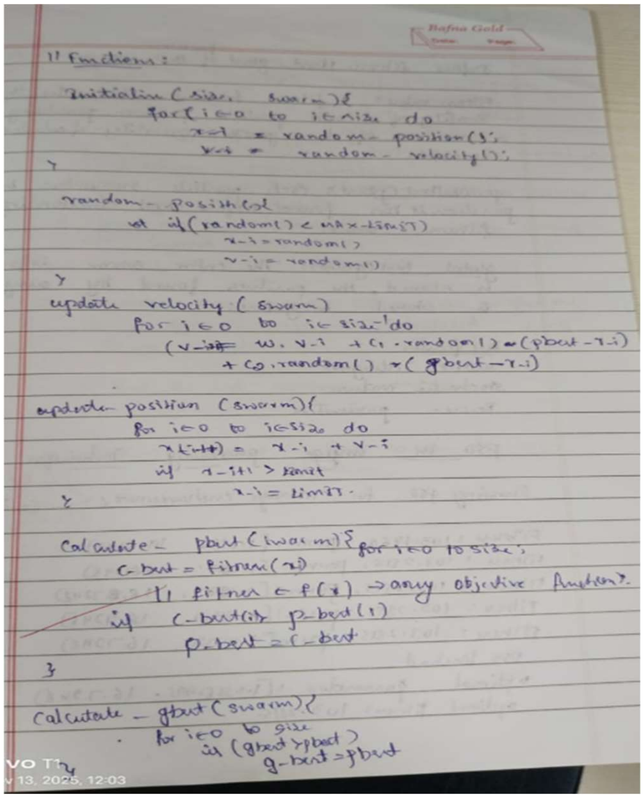
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



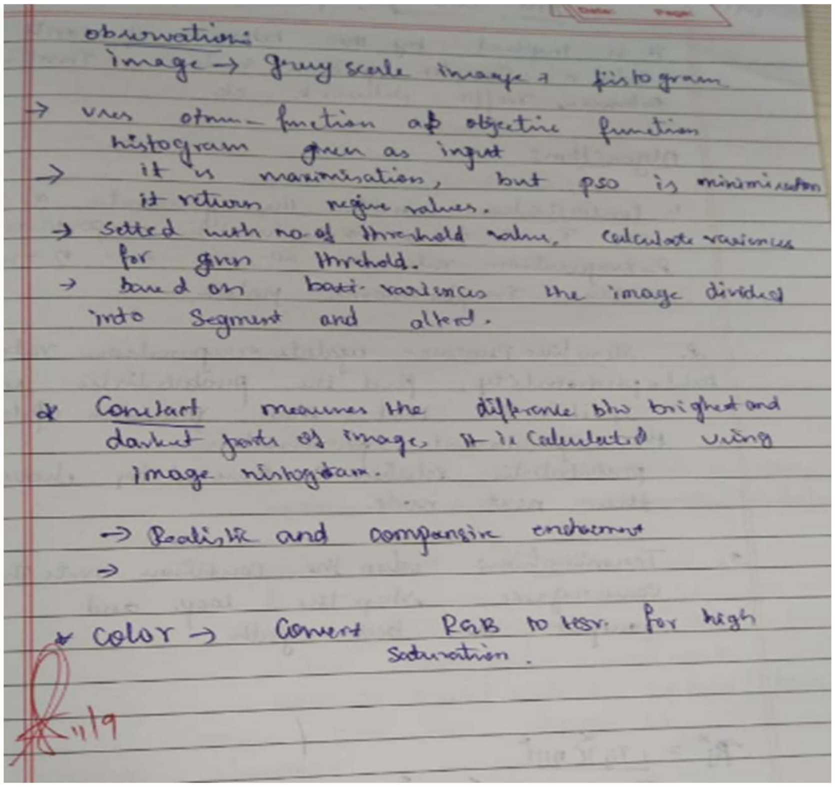
```
Pseudo Code:-

// Main loop.

Initialize (particle
Initialize no of generation n=10
Swarm size =30
Inertia weight = w = 1.0
  Coeficent c1 = 0.4
  Coeficent c2 = 0.6.
Initialize ( size, swarm) //Initialize swarm with
        random velocity & position.

  Set pbest = ~~pe~~personal-best ( swarm) x-i
  Set gbest = the best pbest in the particle swarm

while (n) //termination condition not meet
{
    for  i←0 to i<size  DO
    <    update-velocity (swarm)
         update- position (swarm)
    }    Calculate pbest (Swarm)
         Calculate- gbest (swarm)
    }
  ~~if~~ ( n con
  }  .
  reta
return g.best

}.
```

- Exp. gbest, pbest
- fitness fun'n
- Execute in image Processing



```
// Functions:

 Initialize (size, swarm){
        for(i←0 to i<size do
            x-i = random- position();
            v-i = random- velocity();
 }

 random- position(){
        if(random() < MAX-LIMIT)
              x-i = random()
              v-i = random()
 }
update velocity ( swarm)
        for i←0 to i<size-1 do
          (v-i = w. v-i + c1 . random() * (pbest - x-i)
                + c2 . random() * ( gbest - x-i)

update- position (swarm){
        for i←0 to i<size do
          x(i+1) = x-i + v-i
          if x-i+1 > limit
               x-i = limit.
 }

 Calculate- pbest (swarm){ for i←0 to size;
      c-best = fitness(xi)
          // fitness ← f(x) → any objective function.
      if  c-best(i) > p-best(i)
            p-best = c-best
 }

Calculate - gbest (swarm){
      . for i←0 to size
           if (gbest > pbest)
              g-best = pbest
 }
```

VO T1
13, 2025, 12:03



observation:
image → grey scale image → histogram
→ uses otsu-function as objective function histogram given as input
→ it is maximisation, but pso is minimisation it returns negative values.
→ setted with no of threshold value, calculate variances for given threshold.
→ based on both variances the image divided into segment and altered.

* Contrast measures the difference b/w brightest and darkest parts of image, it is calculated using image histogram.

→ Realistic and comprehensive enhancement
→

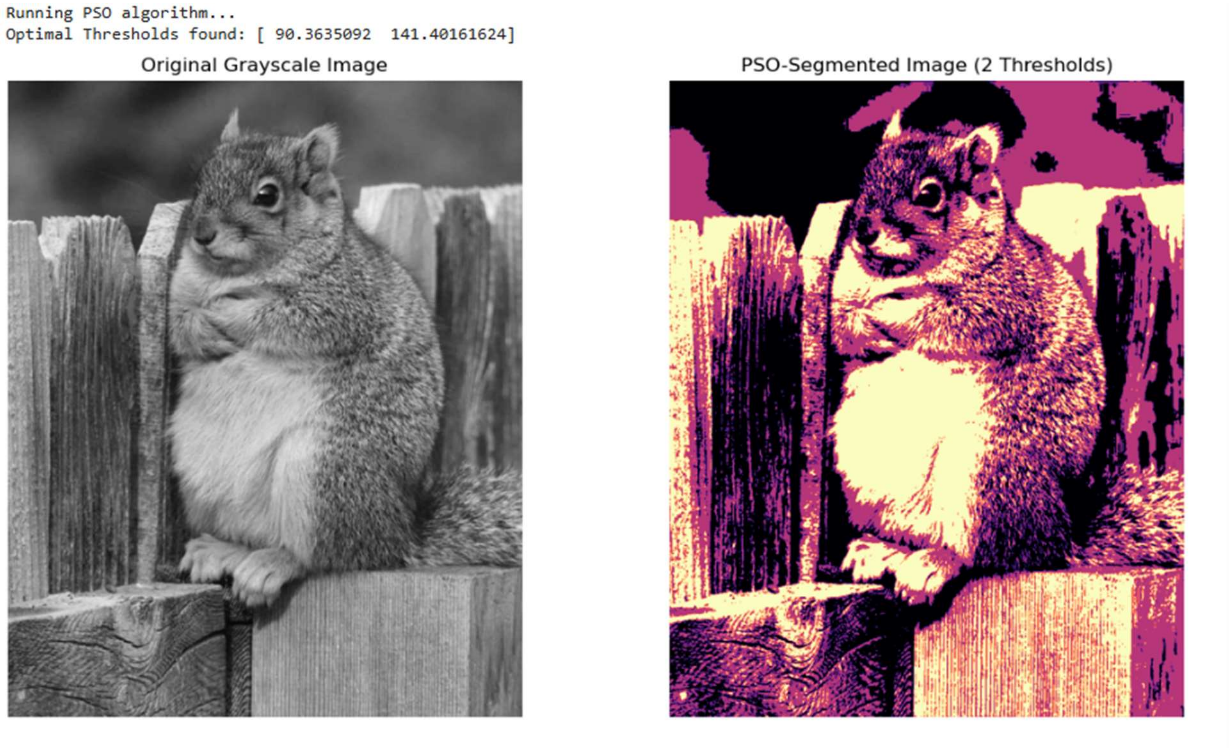
* color → Convert RGB to HSV for high saturation.

11/9

Code:

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

# A simplified representation of a city's road network
# Adjacency matrix where a value indicates the distance between intersections.
# 0 indicates no direct connection.
# Nodes: 0, 1, 2, 3, 4
DISTANCES = np.array([
    [0, 2, 2, 1, 0],
    [2, 0, 1, 0, 3],
    [0, 5, 4, 4, 0],
    [1, 0, 0, 0, 2],
    [0, 0, 0, 2, 0]
])

# Define the initial traffic cost for each road segment (inverse of speed).
# Lower values mean less congestion.
INITIAL_TRAFFIC_COST = np.array([
    [0, 0.2, 0, 0.1, 0],
    [0.2, 0, 0, 0, 0.3],
    [0, 0.5, 0, 0.4, 0],
    [0.1, 0, 0.4, 0, 0.2],
    [0, 0.3, 0, 0.1, 0]
])
```

```
class TrafficACO:
    def __init__(self, distances, traffic_costs, n_ants, n_iterations, decay, alpha=1, beta=2):
        self.distances = distances
        self.traffic_costs = traffic_costs
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta
        self.n_intersections = len(distances)
        self.pheromone = np.ones(self.distances.shape) / self.n_intersections
        self.best_path = None
        self.best_path_cost = float('inf')
        self.best_cost_history = [] # To store the best cost per iteration for plotting

    def run(self, start_node, end_node):
        for i in range(self.n_iterations):
            all_paths = self.generate_paths(start_node, end_node)
            self.update_pheromone(all_paths)

            current_best_cost = float('inf')
            current_best_path = None
            for path in all_paths:
                path_cost = self.calculate_path_cost(path)
                if path_cost < current_best_cost:
                    current_best_cost = path_cost
                    current_best_path = path

            if current_best_cost < self.best_path_cost:
                self.best_path_cost = current_best_cost
                self.best_path = current_best_path

            self.best_cost_history.append(self.best_path_cost)
            print(f"Iteration {i+1}: Best path cost so far = {self.best_path_cost:.2f}")

        return self.best_path, self.best_path_cost

    def generate_paths(self, start_node, end_node):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.construct_path(start_node, end_node)
            all_paths.append(path)
        return all_paths

    def construct_path(self, start_node, end_node):
        path = [start_node]
        current_node = start_node
        visited = {start_node}

        while current_node != end_node:
            next_node = self.choose_next_node(current_node, visited)
            if next_node is None:
                return self.construct_path(start_node, end_node)
            path.append(next_node[0])
            visited.add(next_node[0])
            current_node = next_node[0]

        return path

    def choose_next_node(self, current_node, visited):
        eta = 1.0 / (self.traffic_costs + 1e-10)
        pheromone_weights = self.pheromone[current_node] ** self.alpha
        heuristic_weights = eta[current_node] ** self.beta
        desirability = pheromone_weights * heuristic_weights

        # Avoid already visited nodes and disconnected edges
        desirability[list(visited)] = 0
        desirability[self.distances[current_node] == 0] = 0

        total_desirability = np.sum(desirability)
        if total_desirability == 0:
            return None

        probabilities = desirability / total_desirability
        next_node = random.choices(range(self.n_intersections), weights=probabilities, k=1)
        return next_node

    def update_pheromone(self, all_paths):
        self.pheromone *= (1 - self.decay)
        for path in all_paths:
            path_cost = self.calculate_path_cost(path)
            if path_cost > 0:
                pheromone_deposit = 1.0 / path_cost
                for i in range(len(path) - 1):
                    start, end = path[i], path[i+1]
                    self.pheromone[start, end] += pheromone_deposit
                    self.pheromone[end, start] += pheromone_deposit

    def calculate_path_cost(self, path):
        cost = 0
        for i in range(len(path) - 1):
            start, end = path[i], path[i+1]
            cost += self.distances[start, end] * self.traffic_costs[start, end]
        return cost

# --- Visualization functions ---
def create_and_draw_graph(distances, best_path, start_node, end_node):
    """
    Creates a NetworkX graph and visualizes it.
    The best path is highlighted.
    """
    G = nx.Graph()
    for i in range(len(distances)):
        for j in range(i + 1, len(distances)):
            if distances[i, j] > 0:
                G.add_edge(i, j, weight=distances[i, j])

    pos = nx.spring_layout(G, seed=42)  # For consistent layout
    plt.figure(figsize=(10, 8))

    # Draw all edges in grey
    all_edges = nx.draw_networkx_edges(G, pos, edge_color='gray', width=1)

    # Highlight the best path edges in red with a thicker line
    if best_path:
        path_edges = [(best_path[i], best_path[i+1]) for i in range(len(best_path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='red', width=3)

    # Draw nodes
    nx.draw_networkx_nodes(G, pos, node_color='skyblue', node_size=700)

    # Draw node labels
    nx.draw_networkx_labels(G, pos, font_size=16, font_color='black')

    # Highlight start and end nodes
    nx.draw_networkx_nodes(G, pos, nodelist=[start_node], node_color='green', node_size=800, label="Start")
    nx.draw_networkx_nodes(G, pos, nodelist=[end_node], node_color='purple', node_size=800, label="End")

    plt.title("ACO Traffic Simulation: Best Path Visualization")
    plt.legend(['Start Node', 'End Node'])
    plt.axis('off')
    plt.show()

def plot_cost_over_time(cost_history):
    """
    Plots the best path cost over the course of the iterations.
    """
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(cost_history) + 1), cost_history, marker='o', linestyle='-', color='b')
    plt.title("Best Path Cost Over Iterations")
    plt.xlabel("Iteration")
    plt.ylabel("Lowest Travel Cost")
    plt.grid(True)
    plt.show()

# --- Simulation Execution with Visualization ---
if __name__ == "__main__":
    start_intersection = 0
    end_intersection = 4

    aco = TrafficACO(
        distances=DISTANCES,
        traffic_costs=INITIAL_TRAFFIC_COST,
        n_ants=15,
        n_iterations=33,
        decay=0.5,
        alpha=1,
        beta=2
    )

    best_route, lowest_cost = aco.run(start_node=start_intersection, end_node=end_intersection)

    print("\n--- Results ---")
    print(f"Optimal path: {best_route}")
    print(f"Lowest travel cost: {lowest_cost:.2f}")

    # --- Plotting the results ---
    create_and_draw_graph(DISTANCES, best_route, start_intersection, end_intersection)
    plot_cost_over_time(aco.best_cost_history)
Iteration 1: Best path cost so far = 2.00
Iteration 2: Best path cost so far = 2.00
Iteration 3: Best path cost so far = 2.00
Iteration 4: Best path cost so far = 2.00
Iteration 5: Best path cost so far = 2.00
Iteration 6: Best path cost so far = 2.00
Iteration 7: Best path cost so far = 2.00
Iteration 8: Best path cost so far = 2.00
Iteration 9: Best path cost so far = 2.00
Iteration 10: Best path cost so far = 2.00
Iteration 11: Best path cost so far = 2.00
Iteration 12: Best path cost so far = 2.00
Iteration 13: Best path cost so far = 2.00
Iteration 14: Best path cost so far = 2.00
Iteration 15: Best path cost so far = 2.00
Iteration 16: Best path cost so far = 2.00
Iteration 17: Best path cost so far = 2.00
Iteration 18: Best path cost so far = 2.00
Iteration 19: Best path cost so far = 2.00
Iteration 20: Best path cost so far = 2.00
Iteration 21: Best path cost so far = 2.00
Iteration 22: Best path cost so far = 2.00
Iteration 23: Best path cost so far = 2.00
Iteration 24: Best path cost so far = 2.00
Iteration 25: Best path cost so far = 2.00
Iteration 26: Best path cost so far = 2.00
Iteration 27: Best path cost so far = 2.00
Iteration 28: Best path cost so far = 2.00
Iteration 29: Best path cost so far = 2.00
Iteration 30: Best path cost so far = 2.00
Iteration 31: Best path cost so far = 2.00
Iteration 32: Best path cost so far = 2.00
Iteration 33: Best path cost so far = 2.00

--- Results ---
Optimal path: [0, 2, 3, 4]
Lowest travel cost: 2.00
```



```
Running PSO algorithm...
Optimal Thresholds found: [ 90.3635092  141.40161624]
```

Original Grayscale Image

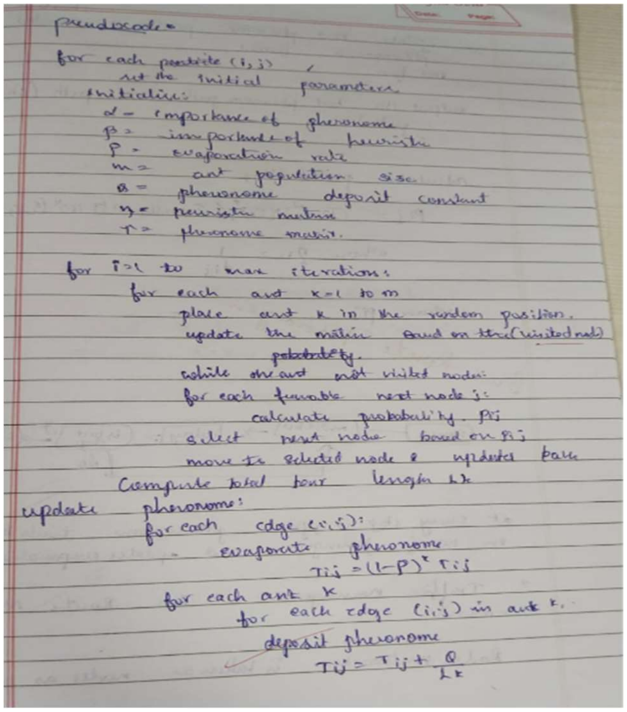
PSO-Segmented Image (2 Thresholds)

Program 3:
Ant Colony Optimization for the Traveling Salesman Problem:

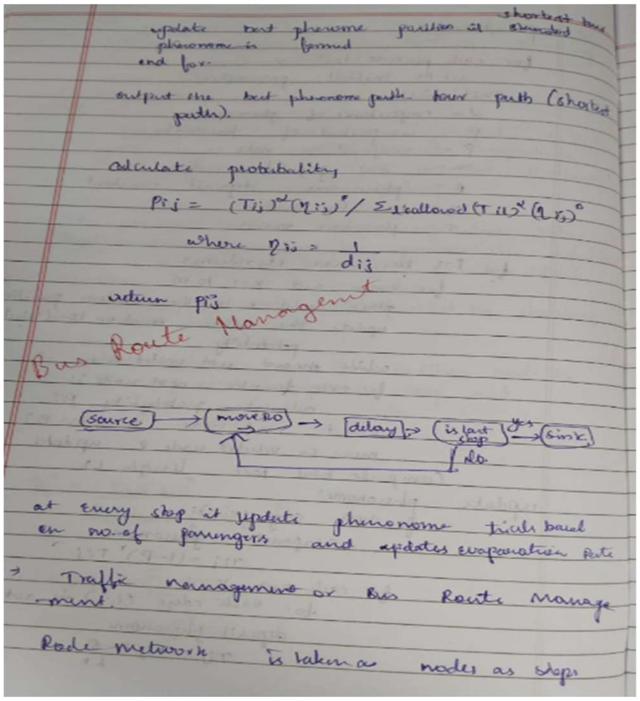
The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm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

Implementation Steps:
1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:



```
Pseudocode:

for each particle (i, j)
    set the initial parameters
initialize:
    α - importance of pheromone
    β = importance of heuristic
    ρ = evaporation rate
    m = ant population size
    Q = pheromone deposit constant
    η = heuristic matrix
    τ = pheromone matrix.

for i=1 to max iterations:
    for each ant k=1 to m
        place ant k in the random position.
        update the matrix based on the (visited nodes)
            probability.
        while the ant not visited nodes:
        for each favorable next node j:
            calculate probability $P_{ij}$
        select next node based on $P_{ij}$
        move to selected node & updates path
    Compute total tour length $L_k$
update pheromone:
    for each edge (i, j):
        evaporate pheromone
            $\tau_{ij} = (1-\rho)^{k} \tau_{ij}$
    for each ant k
        for each edge (i, j) in ant k.
            deposit pheromone
            $\tau_{ij} = \tau_{ij} + \frac{Q}{L_k}$
```



```
    update best pheromone position if shortest tour generated
    pheromone is found
    end for.

    output the best pheromone path tour path (shortest path).

    calculate probability
        $P_{ij} = (\tau_{ij})^{\alpha}(\eta_{ij})^{\beta} / \Sigma_{l \in allowed}(\tau_{il})^{\alpha}(\eta_{il})^{\beta}$
            where $\eta_{ij} = \frac{1}{d_{ij}}$
    return $P_{ij}$
```

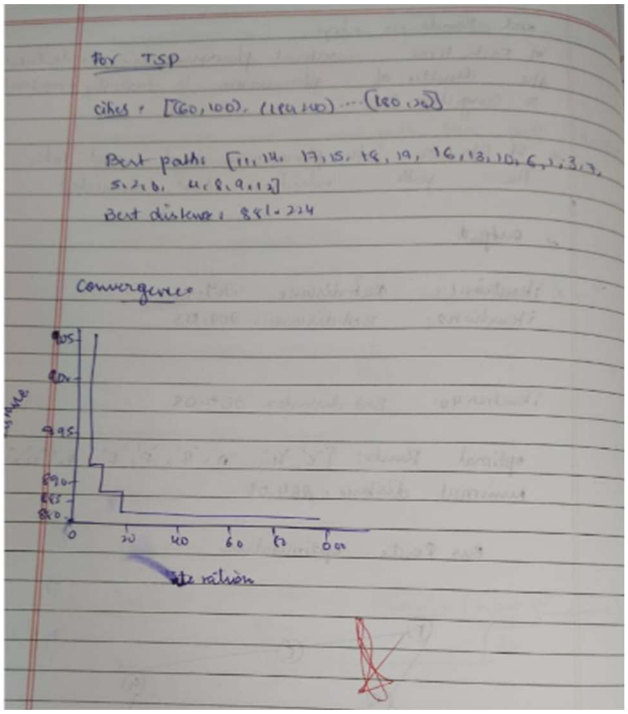
Bus Route Management

Source
moveRO
delay
is last stop
yes
sink
No.

at every stop it updates pheromone trails based on no. of passengers and updates evaporation rate

→ Traffic management or Bus Route Management.

Road network is taken as nodes as stops



For TSP

cities : [(60,100), (184,140) ... (180,20)]

Best path: [11, 14, 17, 15, 18, 19, 16, 13, 10, 6, 1, 3, 7, 5, 2, 0, 4, 8, 9, 12]

Best distance: 881.224

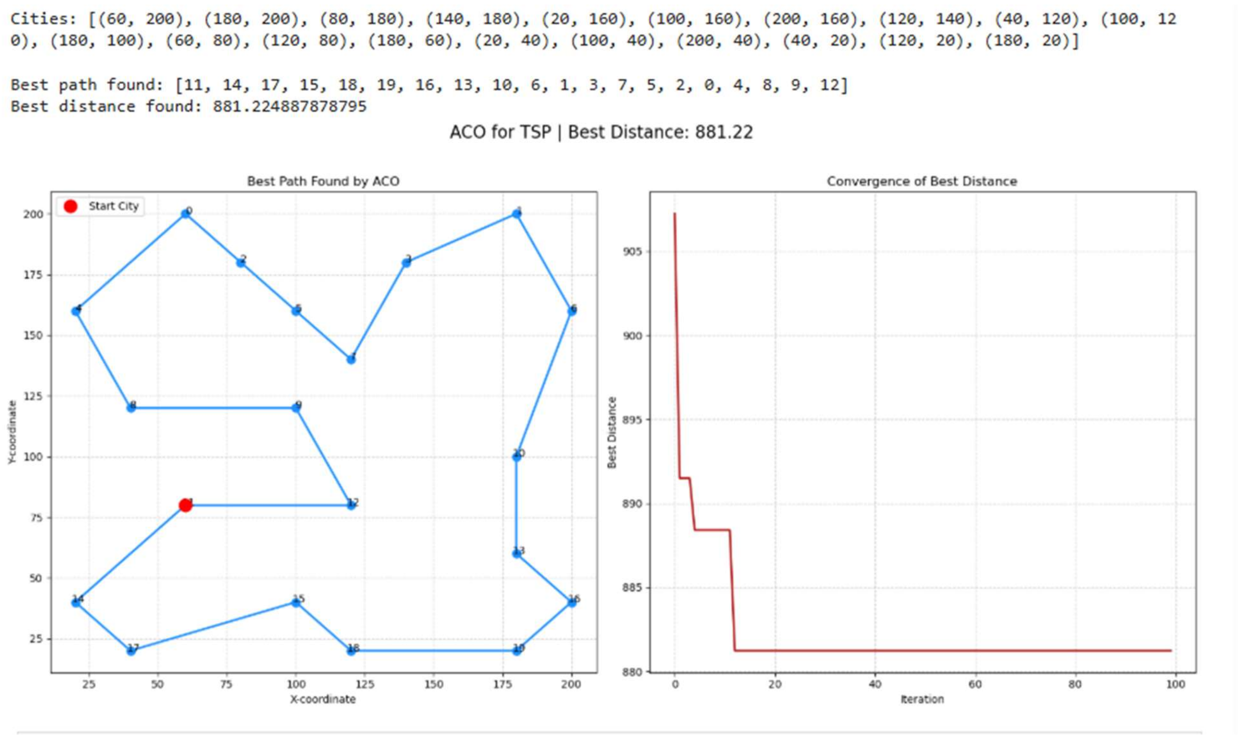
Convergence

Code:

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

#
=====================================================================
=====================
# --- TSP and ACO Classes ---
#
=====================================================================
=====================

class TSP:
    """
    A class to represent the Traveling Salesman Problem (TSP).
    It manages the city coordinates and the pre-computed distance matrix.
    """
    def __init__(self, cities):
        """
        Initializes the TSP problem.
        :param cities: A list of (x, y) tuples representing city coordinates.
        """
        self.cities = cities
        self.num_cities = len(cities)
        self.distance_matrix = self._create_distance_matrix()

    def _create_distance_matrix(self):
        """
        Calculates the Euclidean distance matrix between all pairs of cities.
        :return: A 2D numpy array representing the distances.
        """
        matrix = np.zeros((self.num_cities, self.num_cities))
        for i in range(self.num_cities):
            for j in range(self.num_cities):
                if i != j:
                    matrix[i][j] = self._euclidean_distance(self.cities[i], self.cities[j])
        return matrix

    def _euclidean_distance(self, city1, city2):
        """
        Calculates the Euclidean distance between two cities.
        :param city1: Tuple of (x, y) coordinates for the first city.
        :param city2: Tuple of (x, y) coordinates for the second city.
        :return: The distance between the two cities.
        """
        return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

    def total_distance(self, path):
        """
        Calculates the total distance of a given path.
        :param path: A list of city indices representing a tour.
        :return: The total distance of the tour.
        """
        total = 0
        for i in range(len(path) - 1):
            total += self.distance_matrix[path[i]][path[i+1]]
        total += self.distance_matrix[path[-1]][path[0]]  # Return to start city
        return total

class AntColonyOptimization:
    """
    Implements the Ant Colony Optimization (ACO) algorithm to solve the TSP.
    """
    def __init__(self, problem, num_ants, num_iterations, alpha, beta, evaporation_rate, initial_pheromone):
        """
        Initializes the ACO solver with problem details and parameters.
        :param problem: An instance of the TSP class.
        :param num_ants: Number of ants in the colony.
        :param num_iterations: Number of iterations to run the algorithm.
        :param alpha: Pheromone importance factor.
        :param beta: Heuristic (distance) importance factor.
        :param evaporation_rate: Rate at which pheromones evaporate.
        :param initial_pheromone: Initial amount of pheromone on each path.
        """
        self.problem = problem
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.pheromone = np.full((problem.num_cities, problem.num_cities), initial_pheromone)
        self.best_path = None
        self.best_distance = float('inf')
        self.best_distance_history = []

    def run(self):
        """
        Runs the ACO algorithm for the specified number of iterations.
        :return: The best path and its distance found.
        """
        for _ in range(self.num_iterations):
            all_paths = self._construct_solutions()
            self._update_pheromone(all_paths)
            self._update_best_path(all_paths)
            self.best_distance_history.append(self.best_distance)

        return self.best_path, self.best_distance

    def _construct_solutions(self):
        """
        Constructs a complete tour for each ant in the colony.
        :return: A list of all paths constructed by the ants.
        """
        all_paths = []
        for _ in range(self.num_ants):
            path = self._ant_walk()
            all_paths.append(path)
        return all_paths

    def _ant_walk(self):
        """
        Simulates a single ant constructing a tour.
        :return: A list of city indices representing the constructed path.
        """
        path = []
        start_city = random.randint(0, self.problem.num_cities - 1)
        path.append(start_city)
        visited = {start_city}

        while len(path) < self.problem.num_cities:
            current_city = path[-1]
            unvisited_cities = [city for city in range(self.problem.num_cities) if city not in visited]

            probabilities = self._calculate_probabilities(current_city, unvisited_cities)

            # Select the next city based on probabilities
            next_city = random.choices(unvisited_cities, weights=probabilities, k=1)[0]

            path.append(next_city)
            visited.add(next_city)

        return path
    def _calculate_probabilities(self, current_city, unvisited_cities):
        """
        Calculates the probability of moving to each unvisited city.
        :param current_city: The ant's current city.
        :param unvisited_cities: A list of cities yet to be visited.
        :return: A numpy array of probabilities.
        """
        pheromone_values = np.array([self.pheromone[current_city][city] for city in unvisited_cities])

        # Heuristic values are the inverse of distance
        heuristic_values = np.array([
            1.0 / self.problem.distance_matrix[current_city][city] if
self.problem.distance_matrix[current_city][city] > 0 else 0
            for city in unvisited_cities
        ])

        numerator = (pheromone_values ** self.alpha) * (heuristic_values ** self.beta)
        denominator = np.sum(numerator)

        # Handle case where all probabilities are zero
        if denominator == 0:
            return np.ones(len(unvisited_cities)) / len(unvisited_cities)

        return numerator / denominator

    def _update_pheromone(self, all_paths):
        """
        Updates the pheromone matrix by evaporating and depositing pheromones.
        :param all_paths: A list of all paths from the current iteration.
        """
        # Evaporation
        self.pheromone *= (1 - self.evaporation_rate)

        # Pheromone deposit
        for path in all_paths:
            path_distance = self.problem.total_distance(path)
            pheromone_deposit = 1.0 / path_distance
            for i in range(len(path) - 1):
                self.pheromone[path[i]][path[i+1]] += pheromone_deposit
            self.pheromone[path[-1]][path[0]] += pheromone_deposit

    def _update_best_path(self, all_paths):
        """
        Compares the paths from the current iteration and updates the best path found so far.
        :param all_paths: A list of all paths from the current iteration.
        """
        for path in all_paths:
            distance = self.problem.total_distance(path)
            if distance < self.best_distance:
                self.best_distance = distance
                self.best_path = path
```

```
#
=============================================================
===================
# --- Visualization Function ---
#
=============================================================
===================

def plot_tsp_solution(cities, best_path, best_distance, distance_history):
    """
    Visualizes the optimal path and the convergence of the algorithm.
    :param cities: A list of (x, y) tuples representing city coordinates.
    :param best_path: A list of city indices representing the best tour found.
    :param best_distance: The total distance of the best path.
    :param distance_history: A list of best distances found at each iteration.
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 8))
    fig.suptitle(f'ACO for TSP | Best Distance: {best_distance:.2f}', fontsize=16)

    # Plot the TSP path
    coords = np.array([cities[i] for i in best_path])
    x, y = coords[:, 0], coords[:, 1]

    ax1.plot(x, y, 'o-', markersize=8, color='dodgerblue', linewidth=2)
    ax1.plot([x[-1], x[0]], [y[-1], y[0]], '-', color='dodgerblue', linewidth=2) # Close the loop
    ax1.plot(x[0], y[0], 'ro', markersize=12, label='Start City', zorder=5)

    for i, (cx, cy) in enumerate(cities):
        ax1.text(cx + 1, cy + 1, str(i), fontsize=10, ha='center', va='center')

    ax1.set_title('Best Path Found by ACO')
    ax1.set_xlabel('X-coordinate')
    ax1.set_ylabel('Y-coordinate')
    ax1.grid(True, linestyle='--', alpha=0.6)
    ax1.legend()

    # Plot the convergence of the best distance
    ax2.plot(distance_history, color='firebrick', linewidth=2)
    ax2.set_title('Convergence of Best Distance')
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Best Distance')
    ax2.grid(True, linestyle='--', alpha=0.6)

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

#
=============================================================
===================
# --- Main Execution Block ---
#
=============================================================
====================

if __name__ == '__main__':
    # 1. Define the city coordinates
    cities_list = [
        (60, 200), (180, 200), (80, 180), (140, 180), (20, 160),
        (100, 160), (200, 160), (120, 140), (40, 120), (100, 120),
        (180, 100), (60, 80), (120, 80), (180, 60), (20, 40),
        (100, 40), (200, 40), (40, 20), (120, 20), (180, 20)
    ]

    # 2. Create the TSP problem instance
    tsp_problem = TSP(cities_list)

    # 3. Set ACO parameters
    num_ants = 20
    num_iterations = 100
    alpha = 1.0        # Pheromone importance
    beta = 5.0         # Distance importance (heuristic)
    evaporation_rate = 0.5
    initial_pheromone = 1.0 / (tsp_problem.num_cities * tsp_problem.distance_matrix.mean())

    # 4. Initialize and run the ACO algorithm
    aco_solver = AntColonyOptimization(
        problem=tsp_problem,
        num_ants=num_ants,
        num_iterations=num_iterations,
        alpha=alpha,
        beta=beta,
        evaporation_rate=evaporation_rate,
        initial_pheromone=initial_pheromone
    )
    best_path, best_distance = aco_solver.run()

    # 5. Display and visualize the results
    print("Cities:", cities_list)
    print("\nBest path found:", best_path)
    print("Best distance found:", best_distance)

    # Plot the solution
    plot_tsp_solution(cities_list, best_path, best_distance, aco_solver.best_distance_history)
```

Output:



```
Cities: [(60, 200), (180, 200), (80, 180), (140, 180), (20, 160), (100, 160), (200, 160), (120, 140), (40, 120), (100, 12
0), (180, 100), (60, 80), (120, 80), (180, 60), (20, 40), (100, 40), (200, 40), (40, 20), (120, 20), (180, 20)]

Best path found: [11, 14, 17, 15, 18, 19, 16, 13, 10, 6, 1, 3, 7, 5, 2, 0, 4, 8, 9, 12]
Best distance found: 881.224887878795
```

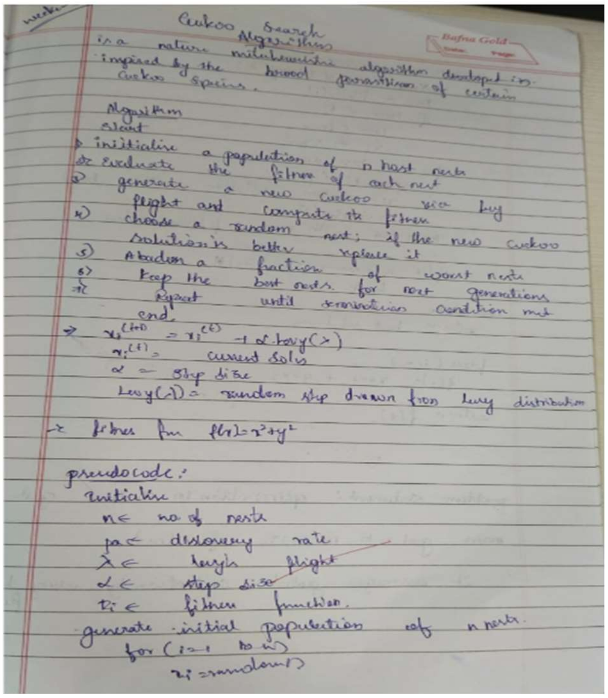
Program 4:

Cuckoo Search (CS):

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

Implementation Steps:
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Cuckoo Search Algorithms

is a nature metaheuristic algorithm developed in inspired by the brood parasitism of certain Cuckoo Species.

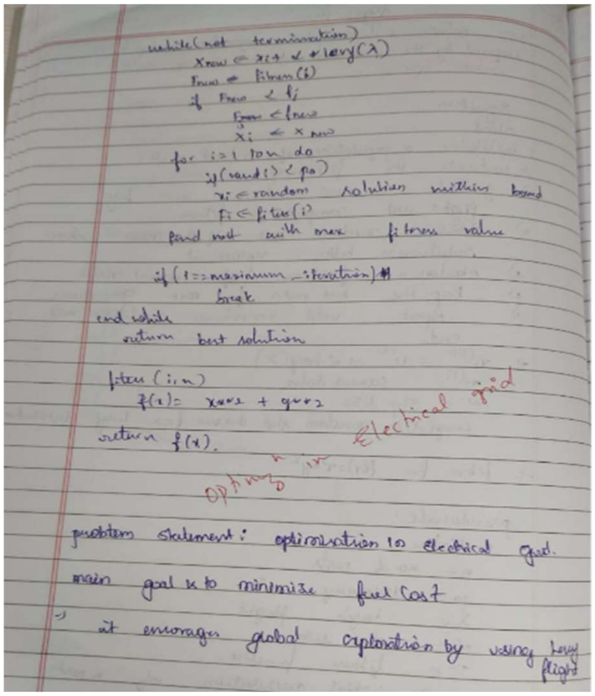
Algorithm
start
1) initialise a population of n host nests
2) Evaluate the fitness of each nest
3) generate a new cuckoo via levy flight and compute its fitness
4) choose a random nest; if the new cuckoo solution is better replace it
5) Abandon a fraction of worst nests
6) Keep the best nests for next generations
7) Repeat until termination condition met
end.

→ $x_i^{(t+1)} = x_i^{(t)} + \alpha \cdot Levy(\lambda)$
$x_i^{(t)}$ = current soln
α = step size
$Levy(\lambda)$ = random step drawn from levy distribution

→ fitness fun $f(x)=x^2+y^2$

pseudocode:

```
initialise
    n ← no of nests
    pa ← discovery rate
    λ ← levy flight
    α ← step size
    fi ← fitness function.
generate initial population of n nests.
    for (i=1 to n)
        xi = random()
```



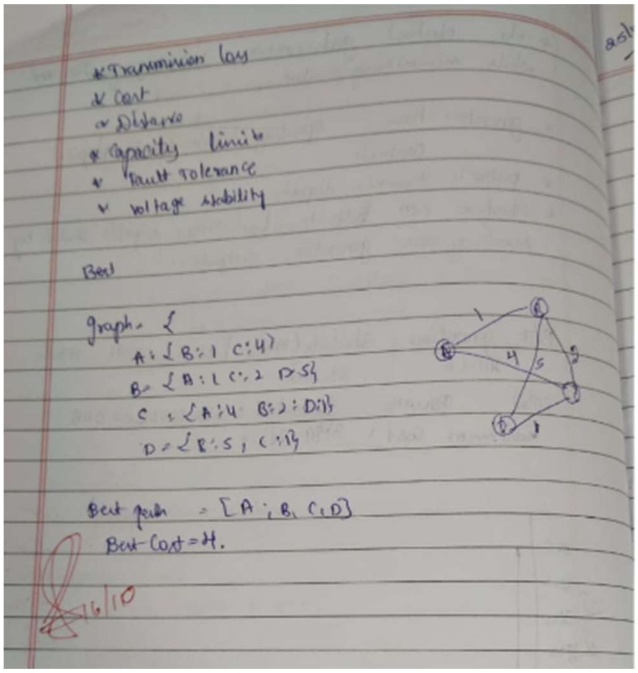
```
while(not termination)
    Xnew ← xi + α * levy(λ)
    Fnew ← fitness(i)
    if Fnew < fi
        fi ← fnew
        xi ← xnew
    for i=1 to n do
        if(rand() < pa)
        xi ← random solution within bound
        fi ← fitness(i)
    find nest with max fitness value

    if (t==maximum_iteration)
        break
end while
    return best solution

fitness (i,n)
    f(x)= x**2 + y**2
return f(x).
```

Optimization in Electrical grid

problem statement: optimisation in electrical grid.

main goal is to minimize fuel cost

→ it encourages global exploration by using levy flight



- Transmision loss
- Cost
- Distance
- Capacity limits
- Fault Tolerance
- voltage stability

Best

graph = {
A : {B : 1 C : 4}
B = {A : 1 C : 2 D : 5}
C : {A : 4 B : 2 : D : 1}
D = {B : 5 , C : 1}

Best path = [A , B, C, D]
Best-Cost = 4.

16/10

Code:

```
import random
# Example graph
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
start_node = 'A'
end_node = 'D'
def path_cost(graph, path):
    cost = 0
    for i in range(len(path) - 1):
        cost += graph[path[i]][path[i+1]]
    return cost
def get_neighbors(graph, node):
    return list(graph[node].keys())
def random_path(graph, start, end):
    """Generate a random valid path from start to end."""
    path = [start]
    current = start
    visited = set(path)
    while current != end:
        neighbors = get_neighbors(graph, current)
        # Exclude already visited nodes to avoid cycles (optional)
        neighbors = [n for n in neighbors if n not in visited]
        if not neighbors:
            # Dead end: restart path generation
            return random_path(graph, start, end)
        current = random.choice(neighbors)
        path.append(current)
        visited.add(current)
    return path
def mutate_path(graph, path):
    """Mutate path by swapping two nodes (except start/end) or inserting a node."""
    if len(path) <= 3:
        # Too short to mutate meaningfully
        return path
    new_path = path.copy()
    # Swap two random internal nodes
    i, j = random.sample(range(1, len(path)-1), 2)
    new_path[i], new_path[j] = new_path[j], new_path[i]
    # Validate mutated path: each consecutive pair must be connected
    for k in range(len(new_path)-1):
        if new_path[k+1] not in graph[new_path[k]]:
            # Invalid path; return original
            return path
    return new_path

def cuckoo_search_path(graph, start, end, n=15, pa=0.25, n_iter=100):
    nests = [random_path(graph, start, end) for _ in range(n)]
    fitness = [path_cost(graph, nest) for nest in nests]

    best_idx = fitness.index(min(fitness))
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]
    fitness_history = [best_fitness]

    for _ in range(n_iter):
        # Generate new solutions by mutation
        for i in range(n):
            new_nest = mutate_path(graph, nests[i])
            new_fitness = path_cost(graph, new_nest)
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness
                if new_fitness < best_fitness:
                    best_nest = new_nest
                    best_fitness = new_fitness

        # Abandon some nests and generate new random ones
        n_abandon = int(pa * n)
        worst_idx = sorted(range(n), key=lambda i: fitness[i], reverse=True)[:n_abandon]
        for idx in worst_idx:
            nests[idx] = random_path(graph, start, end)
            fitness[idx] = path_cost(graph, nests[idx])
            if fitness[idx] < best_fitness:
                best_nest = nests[idx]
                best_fitness = fitness[idx]

        fitness_history.append(best_fitness)

    return best_nest, best_fitness, fitness_history

# Run example
best_path, best_cost, history = cuckoo_search_path(graph, start_node, end_node, n=20, pa=0.3, n_iter=50)
print("Best path:", best_path)
print("Best cost:", best_cost)
```

Output:

```
Best path: ['A', 'B', 'C', 'D']
Best cost: 4
```

Program 5:
Grey Wolf Optimizer (GWO):

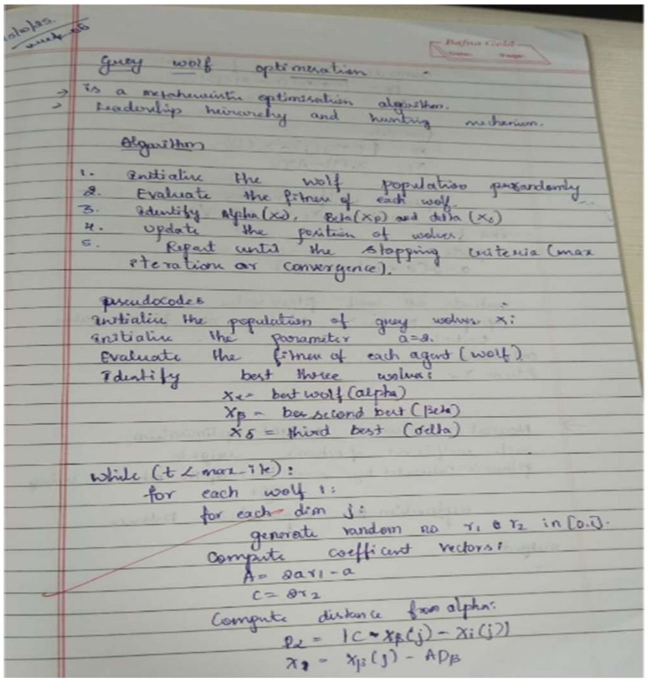
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies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

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.

Output the Best Solution: Track and output the best solution found during the iterations

Algorithm:



10/10/25.
week-06

Grey wolf optimization

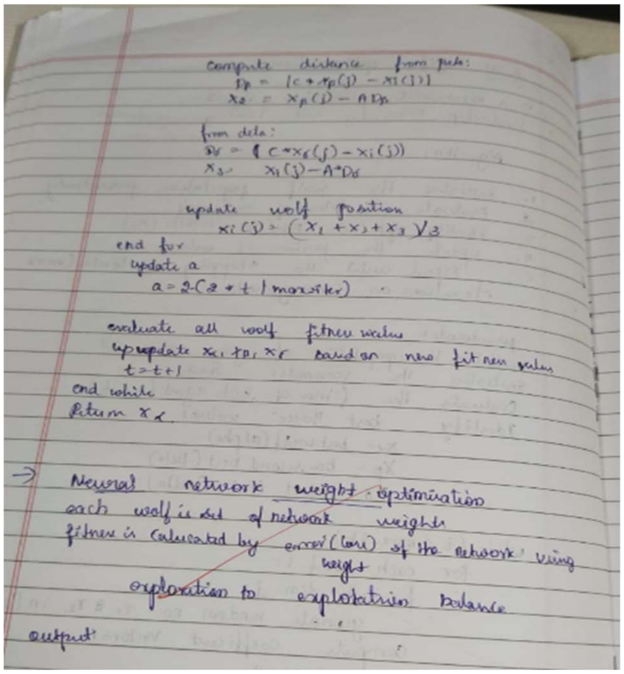
→ is a metaheuristic optimisation algorithm.
→ Leadership hierarchy and hunting mechanism.

Algorithm

1. Initialize the wolf population randomly
2. Evaluate the fitness of each wolf.
3. Identify Alpha (x_α), Beta (x_β) and delta (x_δ)
4. update the position of wolves.
5. Repeat until the stopping criteria (max iteration or convergence).

pseudocodes

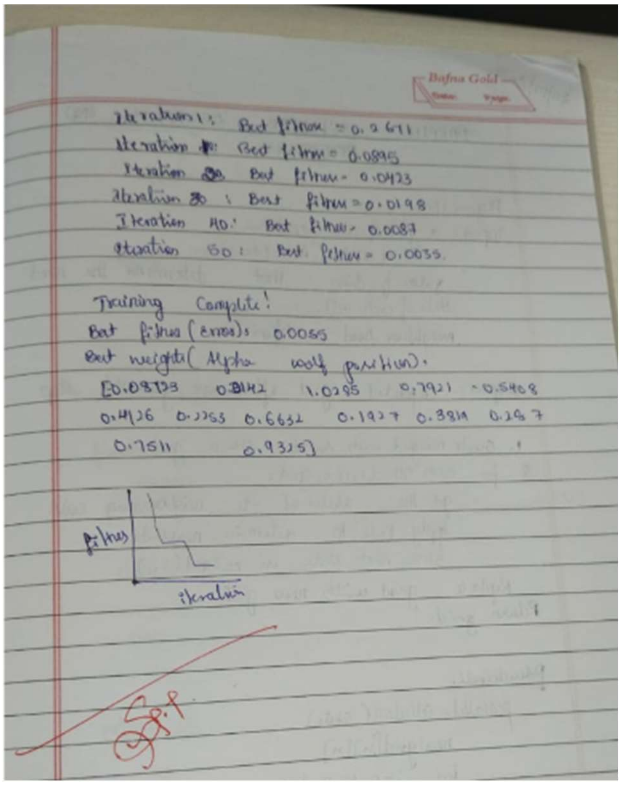
```
Initialize the population of grey wolves x;
Initialize the parameter a=2.
Evaluate the fitness of each agent (wolf)
Identify best three wolves:
        $x_\alpha$ = best wolf (alpha)
        $x_\beta$ = the second best (Beta)
        $x_\delta$ = third best (delta)

while (t < max_ite):
    for each wolf i:
        for each dim j:
            generate random no $r_1$ & $r_2$ in [0,1].
        compute coefficient vectors:
            A = 2a r1 - a
            C = 2r2
        Compute distance from alpha:
            $D_\alpha = |C * x_\beta(j) - x_i(j)|$
            $x_1 = x_\beta(j) - AD_\beta$
```



```
        compute distance from beta:
            $D_\beta = |C * x_\beta(j) - x_i(j)|$
            $x_2 = x_\beta(j) - AD_\beta$

        from delta:
            $D_\delta = |C * x_\delta(j) - x_i(j)|$
            $x_3 = x_1(j) - A*D_\delta$

        update wolf position
            $x_i(j) = (x_1 + x_2 + x_3)/3$
    end for
    update a
        a = 2-(2 * t / max_iter)

    evaluate all wolf fitness values
    update $x_\alpha, x_\beta, x_\delta$ based on new fit new values
    t = t+1
end while
Return $x_\alpha$
```

→ Neural network weight optimization
each wolf is set of network weights
fitness is calculated by error (loss) of the network using weight
exploration to exploitation balance

output:



Iteration 1: Best fitness = 0.2611
Iteration 10: Best fitness = 0.0895
Iteration 20: Best fitness = 0.0423
Iteration 30: Best fitness = 0.0198
Iteration 40: Best fitness = 0.0087
Iteration 50: Best fitness = 0.0035.

Training Complete!
Best fitness (error): 0.0055
Best weights (Alpha wolf position):
[0.08733 0.3142 1.0295 0.7921 -0.5408
0.4126 0.2253 0.6632 0.1927 0.3814 0.2967
0.7511 0.9325]

fitness
iteration

Code:

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Load dataset
data = load_breast_cancer()
X = data.data
y = data.target
num_features = X.shape[1]

# Gray Wolf Optimizer parameters
num_wolves = 10  # Population size
max_iter = 10    # Number of iterations

# Binary GWO helper functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def binary_transform(x):
    return np.where(sigmoid(x) > np.random.rand(len(x)), 1, 0)

# Fitness function: classification accuracy
def fitness(position):
    selected_features = np.where(position == 1)[0]
    if len(selected_features) == 0:
        return 0
    X_selected = X[:, selected_features]
    clf = RandomForestClassifier(n_estimators=50)
    score = cross_val_score(clf, X_selected, y, cv=5).mean()
    return score

# Initialize wolves
wolves = np.random.uniform(-1, 1, (num_wolves, num_features))
binary_wolves = np.array([binary_transform(w) for w in wolves])
fitness_vals = np.array([fitness(w) for w in binary_wolves])

# Initialize alpha, beta, delta
alpha_idx = np.argmax(fitness_vals)
alpha = wolves[alpha_idx].copy()
alpha_score = fitness_vals[alpha_idx]

beta_idx = np.argsort(fitness_vals)[-2]
beta = wolves[beta_idx].copy()
beta_score = fitness_vals[beta_idx]

delta_idx = np.argsort(fitness_vals)[-3]
delta = wolves[delta_idx].copy()
delta_score = fitness_vals[delta_idx]

# Main loop
for t in range(max_iter):
    a = 2 - t * (2 / max_iter)  # Linearly decreasing a

    for i in range(num_wolves):
        for j in range(num_features):
            r1, r2 = np.random.rand(), np.random.rand()
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha[j] - wolves[i][j])
            X1 = alpha[j] - A1 * D_alpha

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

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

            wolves[i][j] = (X1 + X2 + X3) / 3

    # Update binary positions
    binary_wolves = np.array([binary_transform(w) for w in wolves])
    fitness_vals = np.array([fitness(w) for w in binary_wolves])

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

    print(f"Iteration {t+1}/{max_iter}, Best fitness: {alpha_score:.4f}")

# Best feature subset
best_features = np.where(binary_transform(alpha) == 1)[0]
print("Selected feature indices:", best_features)
print("Number of features selected:", len(best_features))
```

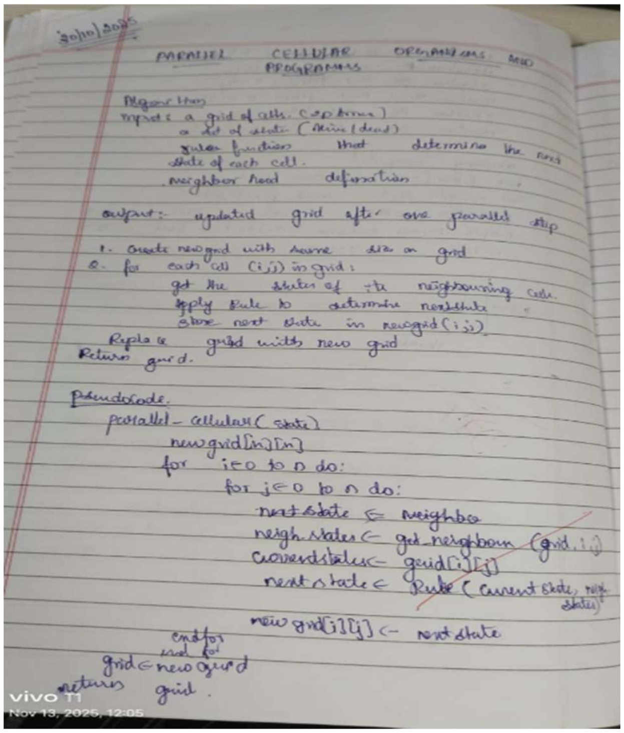
Program 6:

Parallel Cellular Algorithms and Programs:

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

Implementation Steps:
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations. 3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



30/10/2025

PARALLEL CELLULAR ORGANISMS AND PROGRAMS

Algorithm
Inputs: a grid of cells. (n×n form)
a set of state (Alive / dead)
rules function that determine the next state of each cell.
neighbor hood definition

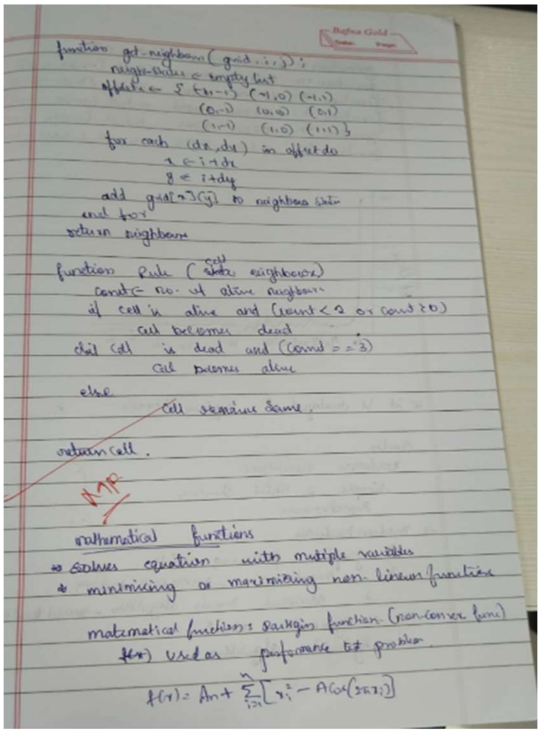
output: updated grid after one parallel step

```
1. Create newgrid with same size as grid
2. for each cell (i,j) in grid:
       get the states of its neighbouring cells.
       apply Rule to determine nextstate
       store next state in newgrid(i,j).
   Replace grid with new grid
Return grid.
```

Pseudocode.

```
parallel_cellular(state)
    newgrid[n][n]
    for i←0 to n do:
        for j←0 to n do:
            next state ⇐ neighbor
            neigh states ← get_neighbours (grid, i, j)
            currentstates ← grid[i][j]
            next state ← Rule (current state, neigh states)

            new grid[i][j] ← next state
        endfor
    end for
    grid ← new grid
return grid.
```

vivo T1
Nov 13, 2025, 12:05



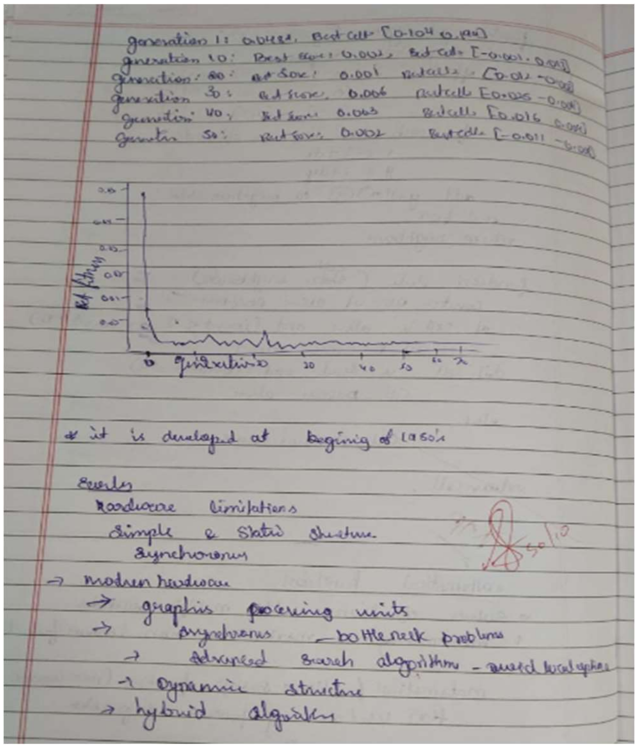
Bafna Gold
Date Page

```
function get_neighbours(grid, i, j);
    neigh-states ← empty list
    offsets ← { (-1,-1) (-1,0) (-1,1)
                (0,-1) (0,0) (0,1)
                (1,-1) (1,0) (1,1) }
    for each (dx, dy) in offset do
        x ← i+dx
        y ← i+dy
        add grid[x][y] to neighbours state
    end for
return neighbours

function Rule (cell state, neighbours)
    count ← no. of alive neighbours
    if cell is alive and (count < 2 or count > 3)
        cell becomes dead
    elif cell is dead and (count == 3)
        cell becomes alive
    else
        cell remains same.

return cell.
```

mathematical functions
- solves equation with multiple variables
- minimizing or maximizing non-linear function

matematical functions: Rastrigin function (non-convex func)
f(x) used as performance test problem.

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)]$$


Best fitness
0.05
0.04
0.03
0.02
0.01
0.0
0
generations
20
40
50
60
x

generation 1: 0.0484, Best cell [0.104 0.[illegible]]
generation 10: Best score: 0.002, Best cell [-0.001, 0.0[illegible]]
generation 20: Best score: 0.001 Best cell: [0.012 -0.008]
generation 30: Best score: 0.006 Best cell [0.025 -0.004]
generation 40: Best score: 0.003 Best cell [0.016 0.008]
Generation 50: Best score: 0.002 Best cell [-0.011 -0.008]

* it is developed at begining of 1950's

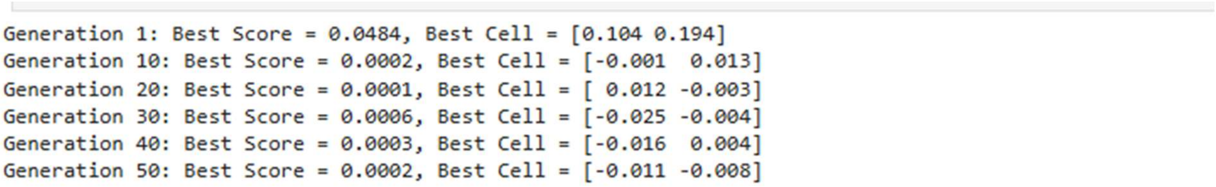
Early
hardware limitations
Simple & Static structure
Synchronous
→ modern hardware
→ graphics processing units
→ Asynchronous – bottleneck problems
→ Advanced search algorithms – guided localsearch
→ Dynamic structure
→ hybrid algorithms

Code:

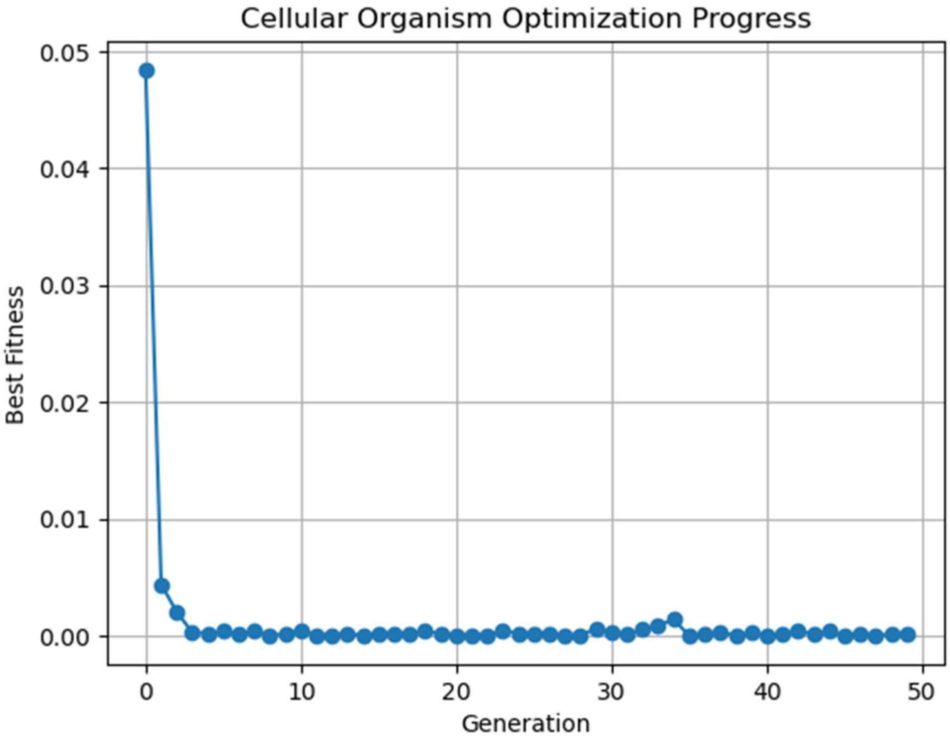
```
import numpy as np
import matplotlib.pyplot as plt

# Objective function to minimize (Sphere function)
def objective_function(x):
    return sum([xi**2 for xi in x])

# Mutation function
def mutate(x, mutation_rate=0.1):
    return [xi + mutation_rate * np.random.randn() for xi in x]

# Update rule: pick best neighbor and mutate
def update_cell(cell, neighbors, mutation_rate=0.1):
    best_neighbor = min(neighbors, key=objective_function)
    new_cell = mutate(best_neighbor, mutation_rate)
    return new_cell

# Initialize grid with random vectors
def initialize_grid(grid_size, dimensions):
    return [[np.random.uniform(-5, 5, dimensions).tolist() for _ in range(grid_size)] for _ in range(grid_size)]

# Get neighbors with wrap-around (toroidal)
def get_neighbors(grid, i, j):
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            ni, nj = (i + di) % len(grid), (j + dj) % len(grid[0])
            if ni != i or nj != j:
                neighbors.append(grid[ni][nj])
    return neighbors

# Evolve grid for one generation
def evolve(grid, mutation_rate=0.1):
    new_grid = []
    for i in range(len(grid)):
        row = []
        for j in range(len(grid[i])):
            neighbors = get_neighbors(grid, i, j)
            new_cell = update_cell(grid[i][j], neighbors, mutation_rate)
            row.append(new_cell)
        new_grid.append(row)
    return new_grid

# Run simulation with step-10 printing
def run_simulation(grid_size=10, dimensions=2, generations=50, mutation_rate=0.1):
    grid = initialize_grid(grid_size, dimensions)
    best_scores = []

    for gen in range(generations):
        grid = evolve(grid, mutation_rate)
       all_cells = [cell for row in grid for cell in row]
        best_cell = min(all_cells, key=objective_function)
        best_score = objective_function(best_cell)
        best_scores.append(best_score)

        # Print every 10 generations
        if (gen + 1) % 10 == 0 or gen == 0:
            print(f"Generation {gen+1}: Best Score = {best_score:.4f}, Best Cell = {np.round(best_cell, 3)}")

    # Plot best score over generations
    plt.plot(best_scores, marker='o', linestyle='-')
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness")
    plt.title("Cellular Organism Optimization Progress")
    plt.grid(True)
    plt.show()

# Run it!
```

run_simulation()

Output:



```
Generation 1: Best Score = 0.0484, Best Cell = [0.104 0.194]
Generation 10: Best Score = 0.0002, Best Cell = [-0.001  0.013]
Generation 20: Best Score = 0.0001, Best Cell = [ 0.012 -0.003]
Generation 30: Best Score = 0.0006, Best Cell = [-0.025 -0.004]
Generation 40: Best Score = 0.0003, Best Cell = [-0.016  0.004]
Generation 50: Best Score = 0.0002, Best Cell = [-0.011 -0.008]
```

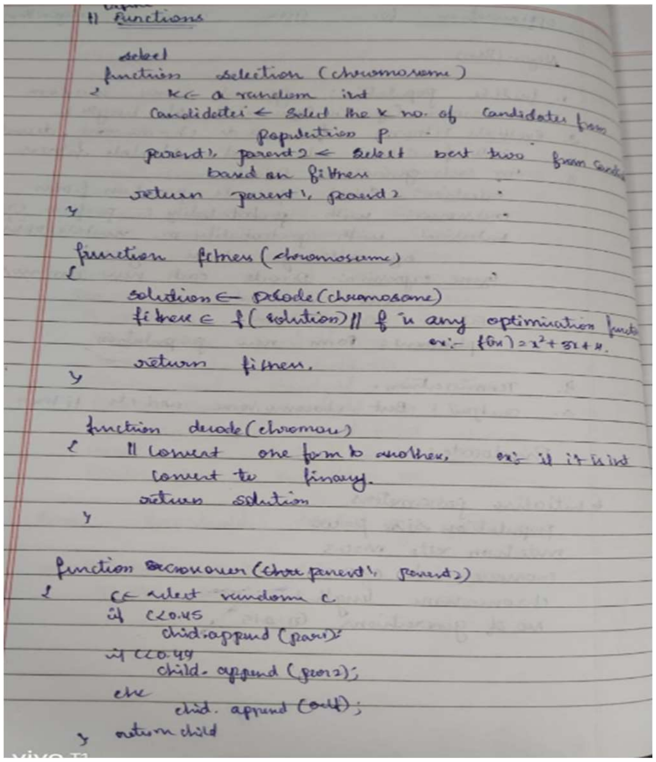


Program 7:

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

Implementation Steps:
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



```
// Functions

  select
  function selection (chromosome)
  {
        k ← a random int
        Candidates ← Select the k no. of candidates from
                     Population P
        parent1, parent2 ← Select best two from candi
                   based on fitness
        return parent1, parent2
  }

function fitness (chromosome)
{
      solution ← decode (chromosome)
      fitness ← f( solution) // f is any optimization functi
                                      ex:- f(x) = x^2 + 3x + 4.
      return fitness.
}

  function decode (chromosome)
  {   // convert one form to another,    ex:- if it is int
          convert to binary.
        return solution
  }

function crossover (chro parent1, parent2)
{     c ← select random c
      if c<0.45
          child.append (par1);
      if c<0.49
          child.append (par2);
      else
          child. append (self);
  }  return child
```

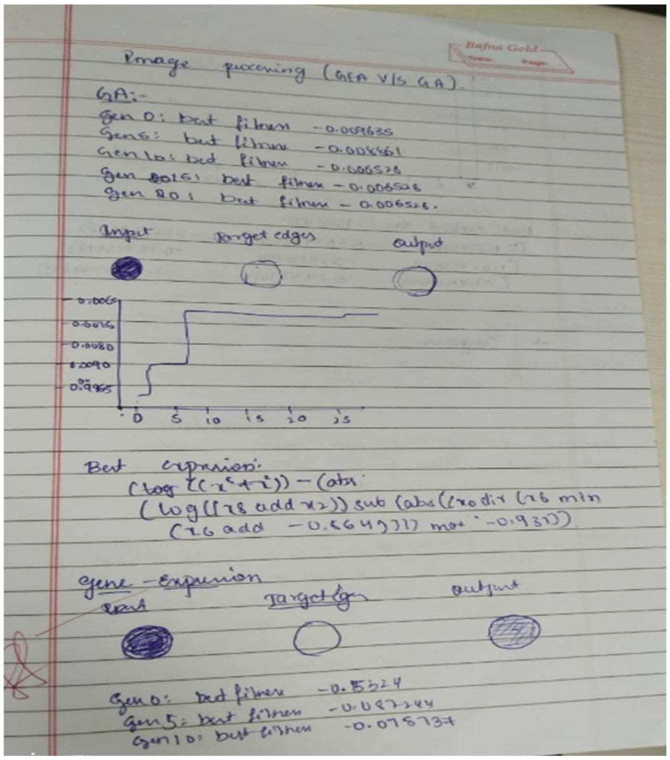


Image processing (GEA V/S GA).

GA:-

Gen 0: best fitness -0.009635
Gen5: best fitness -0.008561
Gen 10: best fitness -0.006526
Gen ~~20~~15: best fitness -0.006526
Gen 20: best fitness -0.006526.

Input
Target edges
output
0.0065
0.0075
0.0080
0.0090
0.0965
0
5
10
15
20
25

Best expression:
(log ((x^2+i)) - (abs
(log((x8 add x2)) sub (abs((x0 div (x6 min
(x6 add -0.5649)))) max -0.937))

Gene-Expression

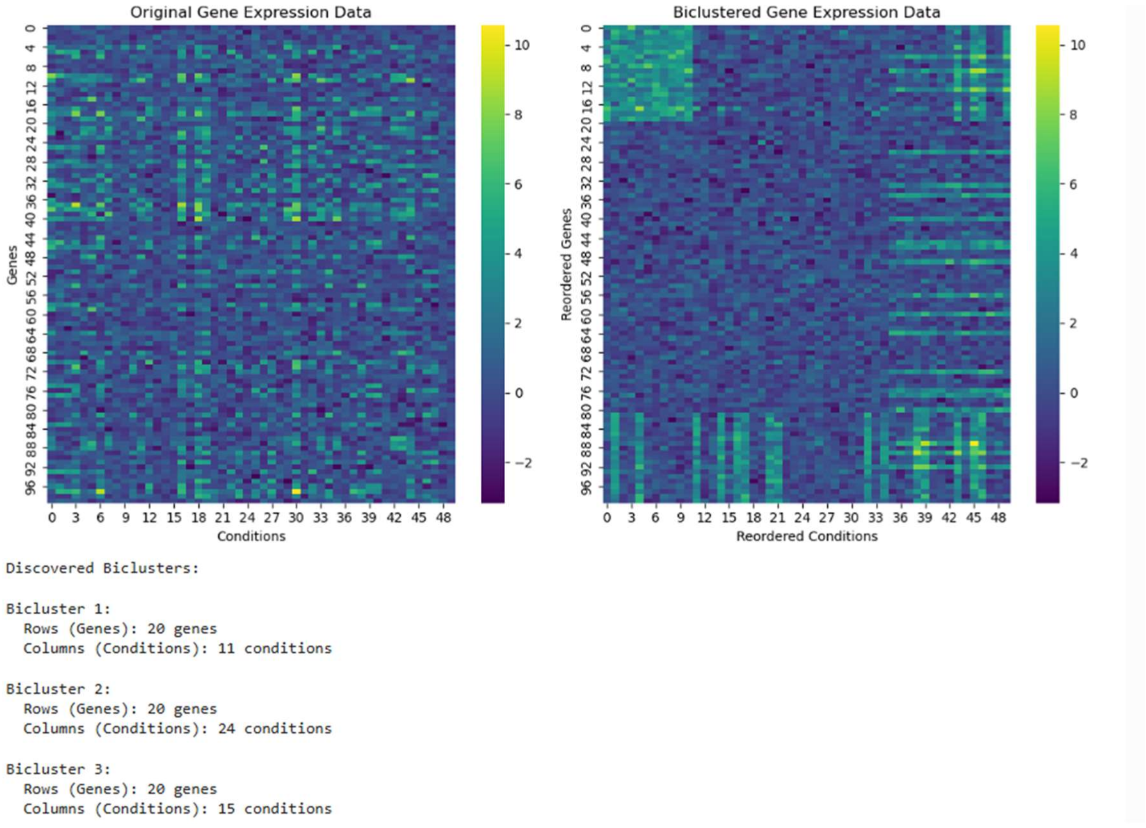
Input
Target Edges
output

Gen 0: best fitness -0.15324
Gen5: best fitness -0.087244
Gen10: best fitness -0.078737

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import SpectralBiclustering

# 1. Generate synthetic gene expression data with known biclusters
# A real-world dataset would be read from a file, e.g., using pandas.
# This code creates a matrix representing genes (rows) and conditions (columns).
n_genes = 100
n_conditions = 50
n_biclusters = 3
noise = 0.5

# Create a base matrix of random noise
np.random.seed(42)
data_matrix = np.random.normal(size=(n_genes, n_conditions))

# Introduce distinct expression patterns for different biclusters
for i in range(n_biclusters):
    # Select random subsets of genes and conditions for each bicluster
    gene_indices = np.random.choice(n_genes, size=20, replace=False)
    cond_indices = np.random.choice(n_conditions, size=15, replace=False)

    # Define a high-expression pattern for the bicluster
    pattern = np.random.uniform(2, 5, size=(len(gene_indices), len(cond_indices)))
    data_matrix[np.ix_(gene_indices, cond_indices)] += pattern

# 2. Apply the biclustering algorithm
# The SpectralBiclustering model is configured to find 3 clusters.
model = SpectralBiclustering(n_clusters=n_biclusters, random_state=42)
model.fit(data_matrix)

# 3. Visualize the results
# Rearrange the matrix based on the biclustering results to show the clusters.
fit_data = data_matrix[np.argsort(model.row_labels_)]
fit_data = fit_data[:, np.argsort(model.column_labels_)]

# Plot the original and biclustered heatmaps
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 6))

sns.heatmap(data_matrix, ax=ax1, cmap="viridis")
ax1.set_title("Original Gene Expression Data")
ax1.set_xlabel("Conditions")
ax1.set_ylabel("Genes")
```

```
sns.heatmap(fit_data, ax=ax2, cmap="viridis")
ax2.set_title("Biclustered Gene Expression Data")
ax2.set_xlabel("Reordered Conditions")
ax2.set_ylabel("Reordered Genes")

plt.tight_layout()
plt.show()
# 4. Interpret the output
# The 'rows_' and 'columns_' attributes of the fitted model indicate the bicluster membership.
print("Discovered Biclusters:")
for i in range(model.n_clusters):
    # Get the row and column indices for each bicluster
    row_bicluster_indices = np.where(model.rows_[i])[0]
    col_bicluster_indices = np.where(model.columns_[i])[0]

    print(f"\nBicluster {i+1}:")
    print(f"  Rows (Genes): {len(row_bicluster_indices)} genes")
    print(f"  Columns (Conditions): {len(col_bicluster_indices)} conditions")
    # Uncomment for detailed gene and condition lists
    # print(f"  Gene Indices: {row_bicluster_indices}")
    # print(f"  Condition Indices: {col_bicluster_indices}")
```

Output:



```
Discovered Biclusters:

Bicluster 1:
  Rows (Genes): 20 genes
  Columns (Conditions): 11 conditions

Bicluster 2:
  Rows (Genes): 20 genes
  Columns (Conditions): 24 conditions

Bicluster 3:
  Rows (Genes): 20 genes
  Columns (Conditions): 15 conditions
```