VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Poorvi Naveen (1BM23CS234)

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
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 **Poorvi Naveen (1BM23CS234),** 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.

Dr. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/PoorviNaveen/Bio-Inspired_Systems.git

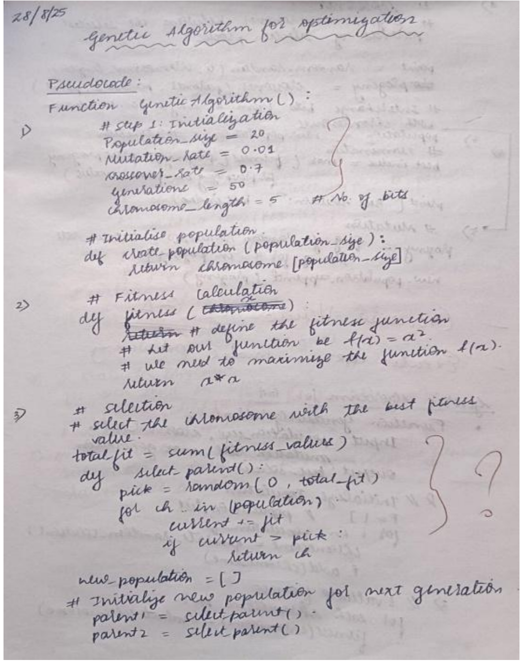
Program 1

Problem statement

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:

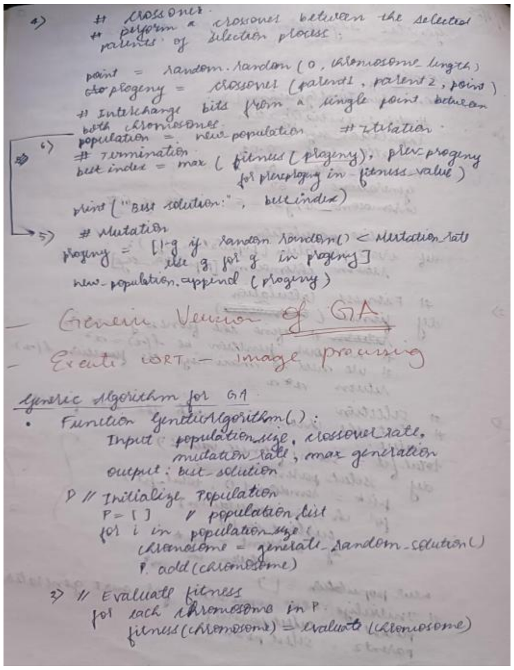


28/8/25

Genetic Algorithm for optimization

Pseudocode:

```
Function Genetic Algorithm () :
1)      # step 1: Initialization
        Population_size = 20
        Mutation_rate = 0.01
        crossover_rate = 0.7
        generations = 50
        chromosome_length = 5     # No. of bits

    # Initialise population
    def create_population (population_size):
            return chromosome [population_size]

2)  # Fitness calculation
    def fitness ( ~~chromosome~~ x ):
        ~~return~~ # define the fitness function
        # let our function be f(x) = x².
        # we need to maximize the function f(x).
        return x*x

3)  # selection
    # select the chromosome with the best fitness
      value.
    total_fit = sum( fitness_values )
    def   select_parent():
        pick = random ( 0 , total_fit )
        for ch in (population)
              current += fit
            if current > pick :
                  return ch

    new_population = [ ]
  # Initialize new population for next generation
    parent1 = select_parent()
    parent2 = select_parent( )
```



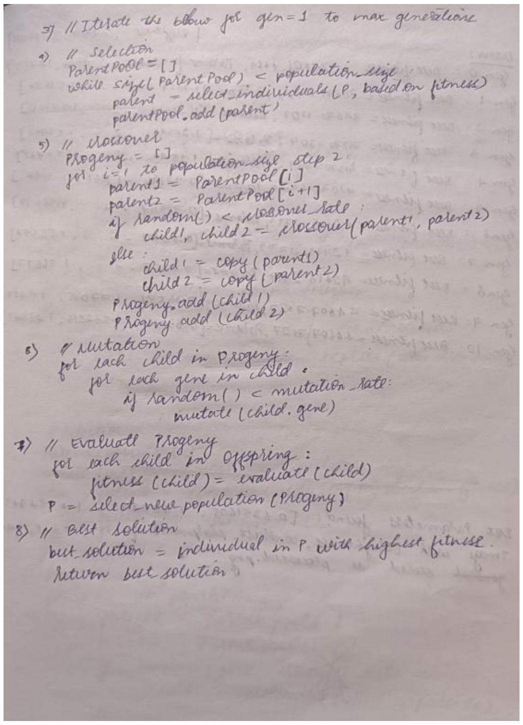
```
4)      # crossover.
        # perform a crossover between the selected
          parents of selection process.

        point = random.random (0, chromosome_length)
        ~~cho~~ progeny = crossover (parent1, parent2, point)
        # Interchange bits from a single point between
          both chromosomes
        population = new_population        # iteration
6)      # Termination
        best index = max ( fitness (progeny), prev_progeny
                           for prevprogeny in fitness_value )

        print ("Best solution:", best index)

5)      # Mutation
        progeny = [1-g if random.random() < Mutation_rate
                        else g for g in progeny]
        new_population.append (progeny)
```

Generic Version of GA

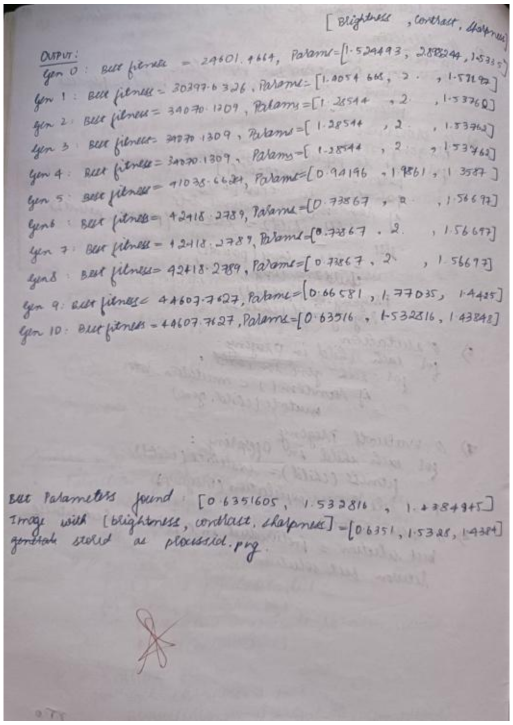
Execute WORT — image processing

Generic Algorithm for GA

```
•   Function GeneticAlgorithm( ):
        Input : population_size, crossover_rate,
                mutation_rate, max generation
        Output : best_solution
    1) // Initialize Population
        P = [ ]     // population list
        for i in population_size :
            chromosome = generate_random_solution( )
            P. add (chromosome)

    2) // Evaluate fitness
        for each chromosome in P
            fitness(chromosome) = evaluate (chromosome)
```



```
3) // Iterate the below for gen = 1 to max generations
4) // Selection
   ParentPool = []
   while size(ParentPool) < population_size
         parent = select_individuals(P, based on fitness)
         parentPool.add(parent)
5) // crossover
   Progeny = []
   for i=1 to population_size step 2
         parent1 = ParentPool[i]
         parent2 = ParentPool[i+1]
         if random() < crossover_rate :
               child1, child2 = crossover(parent1, parent2)
         else :
               child1 = copy(parent1)
               child2 = copy(parent2)
         Progeny.add(child1)
         Progeny.add(child2)
6) // Mutation
   for each child in Progeny:
       for each gene in child :
           if random() < mutation_rate:
                 mutate(child.gene)
7) // Evaluate Progeny
   for each child in Offspring :
         fitness(child) = evaluate(child)
   P = select_new_population(Progeny)
8) // Best solution
   best_solution = individual in P with highest fitness.
   Return best_solution
```



```
                                            [Brightness , Contrast, Sharpness]
OUTPUT:
gen 0 : Best fitness = 24601.4664, Params=[1.524493, 2.898244, 1.5335]
gen 1 : Best fitness = 30397.6326, Params=[1.4054665, 2 ,  1.57197]
gen 2 : Best fitness = 34070.1209, Params=[1.28544  , 2 ,  1.53760]
gen 3 : Best fitness = 34070.1309, Params=[1.28544  , 2 ,  1.53762]
gen 4 : Best fitness = 34070.1309, Params=[1.28544  , 2 ,  1.53762]
gen 5 : Best fitness = 41038.6624, Params=[0.94196 , 1.9861 , 1.3587]
gen6  : Best fitness = 42418.2789, Params=[0.73867 , 2 ,  1.56697]
gen 7 : Best fitness = 42418.2789, Params=[0.73867 , 2 ,  1.56697]
gen8  : Best fitness = 42418.2789, Params=[0.73867 , 2 ,  1.56697]
gen 9 : Best fitness = 44607.7627, Params=[0.66581 , 1.77035, 1.4425]
gen 10: Best fitness = 44607.7627, Params=[0.63516 , 1.532816, 1.43848]

Best Parameters found [0.6351605, 1.532816 , 1.4384945]
Image with [brightness, contrast, sharpness] = [0.6351, 1.5328, 1.4384]
generate stored as processed.png
```

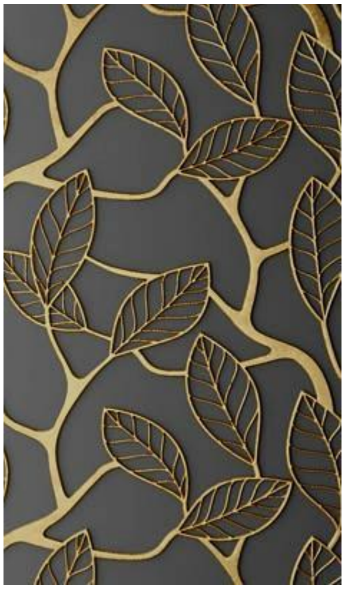
Code:

```
import random
import numpy as np
from PIL import Image, ImageEnhance
from skimage.metrics import structural_similarity as ssim
from skimage.filters import sobel
from skimage import img_as_float
# ---------- Parameters ----------
POP_SIZE = 20
N_GEN = 15
MUT_RATE = 0.3
ELITE = 2

# Load image
original = Image.open("myImage.jpg").convert("RGB")
original_np = np.array(original)

# For a reference image, load it
try:
    target = Image.open("target.jpg").convert("RGB")
    target_np = np.array(target)
    USE_REFERENCE = True
except:
    USE_REFERENCE = False

# ---------- Apply Adjustments ----------
def apply_adjustments(params):
    b, c, s = params
    img = original.copy()
    img = ImageEnhance.Brightness(img).enhance(b)
    img = ImageEnhance.Contrast(img).enhance(c)
    img = ImageEnhance.Sharpness(img).enhance(s)
    return img

# ---------- Fitness Function ----------
def fitness(params):
    img = apply_adjustments(params)
    img_np = np.array(img)

    if USE_REFERENCE:
        return ssim(target_np, img_np, channel_axis=2)
    else:
        gray = img.convert("L")
        gray_np = img_as_float(np.array(gray))
        entropy = -np.sum(gray_np * np.log2(gray_np + 1e-10))
        edge_strength = np.mean(sobel(gray_np))
        return entropy + edge_strength

# ---------- GA Core ----------
def init_population(size):
    return [np.array([
        random.uniform(1.5, 3.5),  # Brightness
        random.uniform(1.5, 3.0),  # Contrast
        random.uniform(1.0, 2.0)   # Sharpness
    ]) for _ in range(size)]

def selection(pop, fitnesses):
    i, j = random.sample(range(len(pop)), 2)
    return pop[i] if fitnesses[i] > fitnesses[j] else pop[j]

def crossover(p1, p2):
    alpha = random.random()
    return alpha * p1 + (1 - alpha) * p2

def mutation(ind):
    if random.random() < MUT_RATE:
        ind += np.random.normal(0, 0.2, size=3)
        ind = np.clip(ind, 0.5, 2.0)
    return ind

# ---------- Run GA ----------
def run_ga():
    pop = init_population(POP_SIZE)

    for gen in range(N_GEN):
        fitnesses = [fitness(ind) for ind in pop]
        ranked = sorted(zip(pop, fitnesses), key=lambda x: x[1], reverse=True)
        best_ind, best_fit = ranked[0]
        print(f"Gen {gen}: Best fitness = {best_fit:.4f}, Params = {best_ind}")

        new_pop = [ind.copy() for ind, _ in ranked[:ELITE]]

        while len(new_pop) < POP_SIZE:
            p1, p2 = select(pop, fitnesses), select(pop, fitnesses)
            child = crossover(p1, p2)
            child = mutate(child)
            new_pop.append(child)
        # new generation
        pop = new_pop

    best_img = apply_adjustments(best_ind)
    best_img.save("processed.jpg")
    print("Best parameters found:", best_ind)
    print(best_img)

run_ga()
print("\n\nExceuted by Poorvi Naveen")
```

<u>Output:</u>

myImage.jpg

processed.jpg

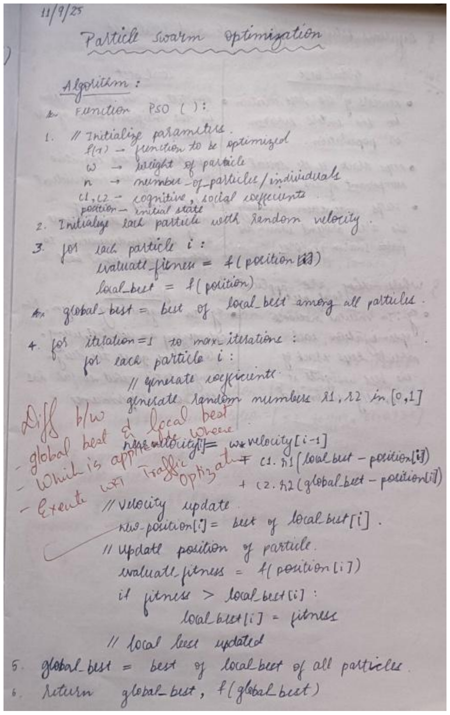


Program 2

Problem statement

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

Algorithm:



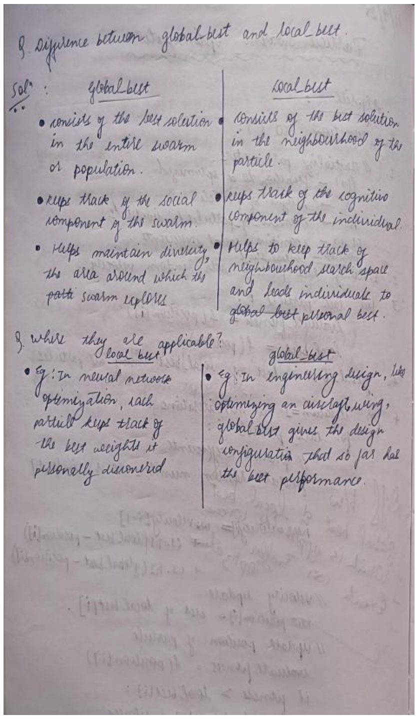
11/9/25

Particle Swarm Optimization

Algorithm :

```
Function PSO ( ):
1. // Initialize parameters.
      f(x) → function to be optimized
      ω → weight of particle
      n → number of particles / individuals
      c1, c2 → cognitive, social coefficients
      position → initial state
2. Initialize each particle with random velocity.
3. for each particle i :
      evaluate_fitness = f(position[i])
      local_best = f(position)
   global_best = best of local_best among all particles.
4. for iteration = 1 to max_iterations :
      for each particle i :
         // generate coefficients
         generate random numbers r1, r2 in [0,1]
         new velocity[i] = w * velocity[i-1]
                           + c1 . r1 (local_best - position[i])
                           + c2 . r2 (global_best - position[i])
         // velocity update
         new-position[i] = best of local best[i] .
         // update position of particle.
         evaluate fitness = f(position[i])
         if fitness > local_best[i] :
               local best[i] = fitness
      // local best updated
5. global_best = best of local best of all particles.
6. return global_best, f(global_best)
```

Diff b/w global best & local best
- Which is applicable where
- Execute wrt Traffic Optimization



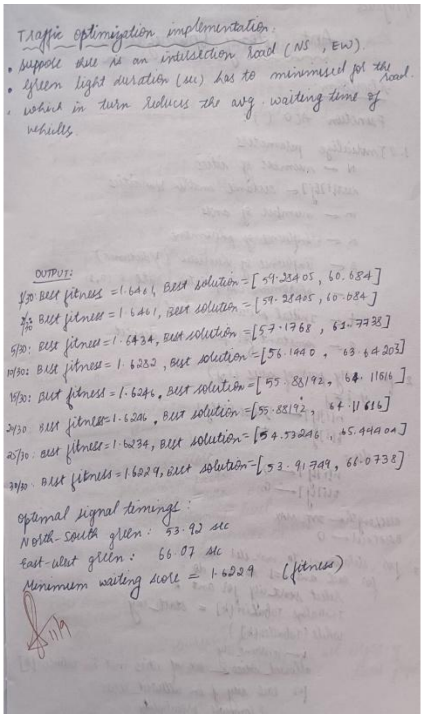
Q. Difference between global_best and local_best.

Sol^n :

global best	local best
• consists of the best solution in the entire swarm or population.	• consists of the best solution in the neighbourhood of the particle.
• keeps track of the social component of the swarm.	• keeps track of the cognitive component of the individual.
• Helps maintain diversity, the area around which the ~~path~~ swarm explores	• Helps to keep track of neighbourhood search space and leads individuals to ~~global best~~ personal best.

Q. where they are applicable?

local best	global best
• Eg: In neural network optimization, each particle keeps track of the best weights it personally discovered	• Eg: In engineering design, like optimizing an aircraft wing, global best gives the design configuration that so far has the best performance.



Traffic optimization implementation:
- Suppose there is an intersection road (NS , EW).
- Green light duration (sec) has to minimised for the road.
- which in turn reduces the avg. waiting time of vehicles.

```
OUTPUT:
1/30: Best fitness = 1.6461, Best solution = [59.28405, 60.684]
2/30: Best fitness = 1.6461, Best solution = [59.28405, 60.684]
5/30: Best fitness = 1.6434, Best solution = [57.1768 , 61.7738]
10/30: Best fitness = 1.6282 , Best solution = [56.1440 , 63.64203]
15/30: Best fitness = 1.6246, Best solution = [55.88192, 64.11616]
20/30: Best fitness = 1.6246 , Best solution = [55.88192, 64.11616]
25/30: Best fitness = 1.6234, Best solution = [54.53246 , 65.44904]
30/30: Best fitness = 1.6229, Best solution = [53.91749, 66.0738]

Optimal signal timings :
North-South green : 53.92 sec
East-West green : 66.07 sec
Minimum waiting score = 1.6229 (fitness)
```

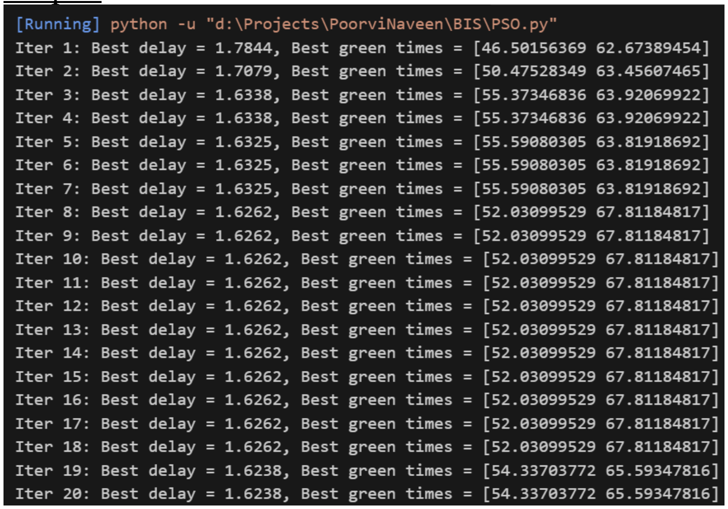
11/9

Code:

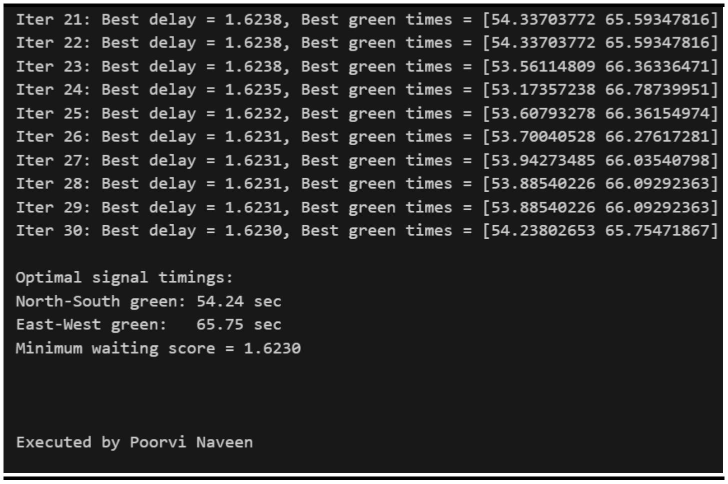
```
import numpy as np
def traffic_delay(position):
    green_NS, green_EW = position
    cycle = 120  # total cycle length in seconds
    if green_NS + green_EW > cycle:
        return 1e6
    lambda_NS, lambda_EW = 40, 60
    delay = (lambda_NS / (green_NS + 1)) + (lambda_EW / (green_EW + 1))
    return delay

# --- PSO parameters ---
n_particles = 10
n_iterations = 30
w, c1, c2 = 0.7, 1.5, 1.5  # inertia, cognitive, social
lb, ub = np.array([10, 10]), np.array([110, 110])

positions = np.random.uniform(lb, ub, (n_particles, 2))
velocities = np.random.uniform(-1, 1, (n_particles, 2))
pbest = positions.copy()
pbest_val = np.array([traffic_delay(p) for p in positions])
```

```
gbest = pbest[np.argmin(pbest_val)]
gbest_val = np.min(pbest_val)
for it in range(n_iterations):
    for i in range(n_particles):
        r1, r2 = np.random.rand(2)
        velocities[i] = (w * velocities[i]
                        + c1 * r1 * (pbest[i] - positions[i])
                        + c2 * r2 * (gbest - positions[i]))
        positions[i] = np.clip(positions[i] + velocities[i], lb, ub)
        val = traffic_delay(positions[i])
        if val < pbest_val[i]:
            pbest[i], pbest_val[i] = positions[i].copy(), val
    if np.min(pbest_val) < gbest_val:
        gbest, gbest_val = pbest[np.argmin(pbest_val)], np.min(pbest_val)
    print(f"Iter {it+1}: Best delay = {gbest_val:.4f}, Best green times = {gbest}")

print("\nOptimal signal timings:")
print(f"North-South green: {gbest[0]:.2f} sec")
print(f"East-West green:   {gbest[1]:.2f} sec")
print(f"Minimum waiting score = {gbest_val:.4f}")
print(“\n\nExceuted by Poorvi Naveen”)
```

Output:



```
[Running] python -u "d:\Projects\PoorviNaveen\BIS\PSO.py"
Iter 1: Best delay = 1.7844, Best green times = [46.50156369 62.67389454]
Iter 2: Best delay = 1.7079, Best green times = [50.47528349 63.45607465]
Iter 3: Best delay = 1.6338, Best green times = [55.37346836 63.92069922]
Iter 4: Best delay = 1.6338, Best green times = [55.37346836 63.92069922]
Iter 5: Best delay = 1.6325, Best green times = [55.59080305 63.81918692]
Iter 6: Best delay = 1.6325, Best green times = [55.59080305 63.81918692]
Iter 7: Best delay = 1.6325, Best green times = [55.59080305 63.81918692]
Iter 8: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 9: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 10: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 11: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 12: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 13: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 14: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 15: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 16: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 17: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 18: Best delay = 1.6262, Best green times = [52.03099529 67.81184817]
Iter 19: Best delay = 1.6238, Best green times = [54.33703772 65.59347816]
Iter 20: Best delay = 1.6238, Best green times = [54.33703772 65.59347816]
```



```
Iter 21: Best delay = 1.6238, Best green times = [54.33703772 65.59347816]
Iter 22: Best delay = 1.6238, Best green times = [54.33703772 65.59347816]
Iter 23: Best delay = 1.6238, Best green times = [53.56114809 66.36336471]
Iter 24: Best delay = 1.6235, Best green times = [53.17357238 66.78739951]
Iter 25: Best delay = 1.6232, Best green times = [53.60793278 66.36154974]
Iter 26: Best delay = 1.6231, Best green times = [53.70040528 66.27617281]
Iter 27: Best delay = 1.6231, Best green times = [53.94273485 66.03540798]
Iter 28: Best delay = 1.6231, Best green times = [53.88540226 66.09292363]
Iter 29: Best delay = 1.6231, Best green times = [53.88540226 66.09292363]
Iter 30: Best delay = 1.6230, Best green times = [54.23802653 65.75471867]

Optimal signal timings:
North-South green: 54.24 sec
East-West green:   65.75 sec
Minimum waiting score = 1.6230



Executed by Poorvi Naveen
```

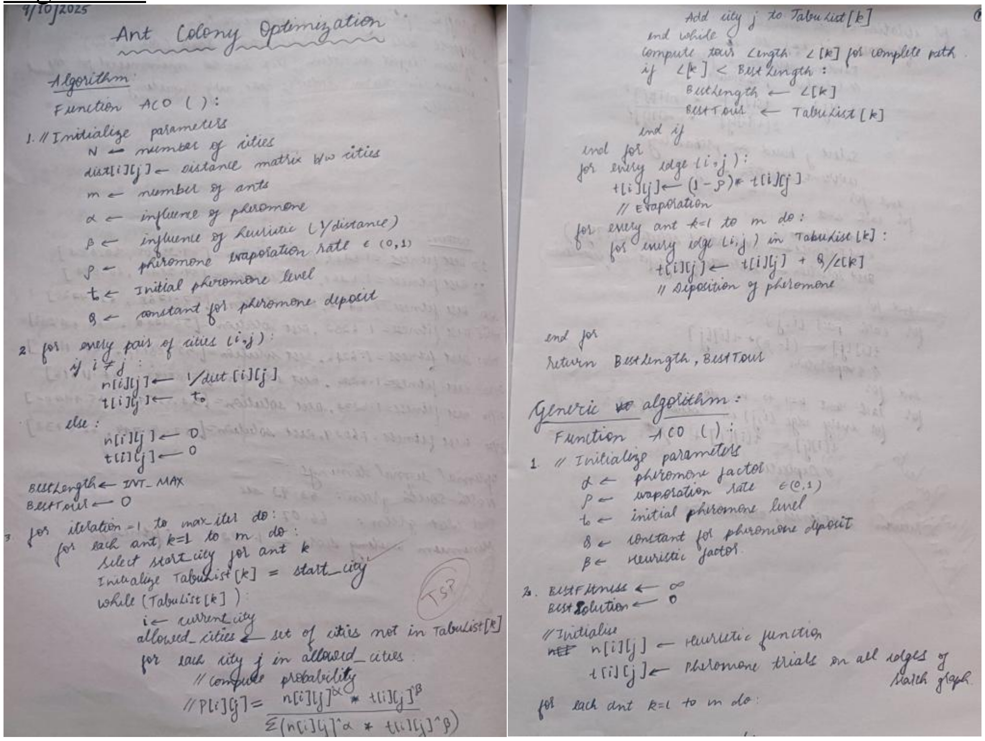
Program 3

Problem statement

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:



9/10/2025

Ant Colony Optimization

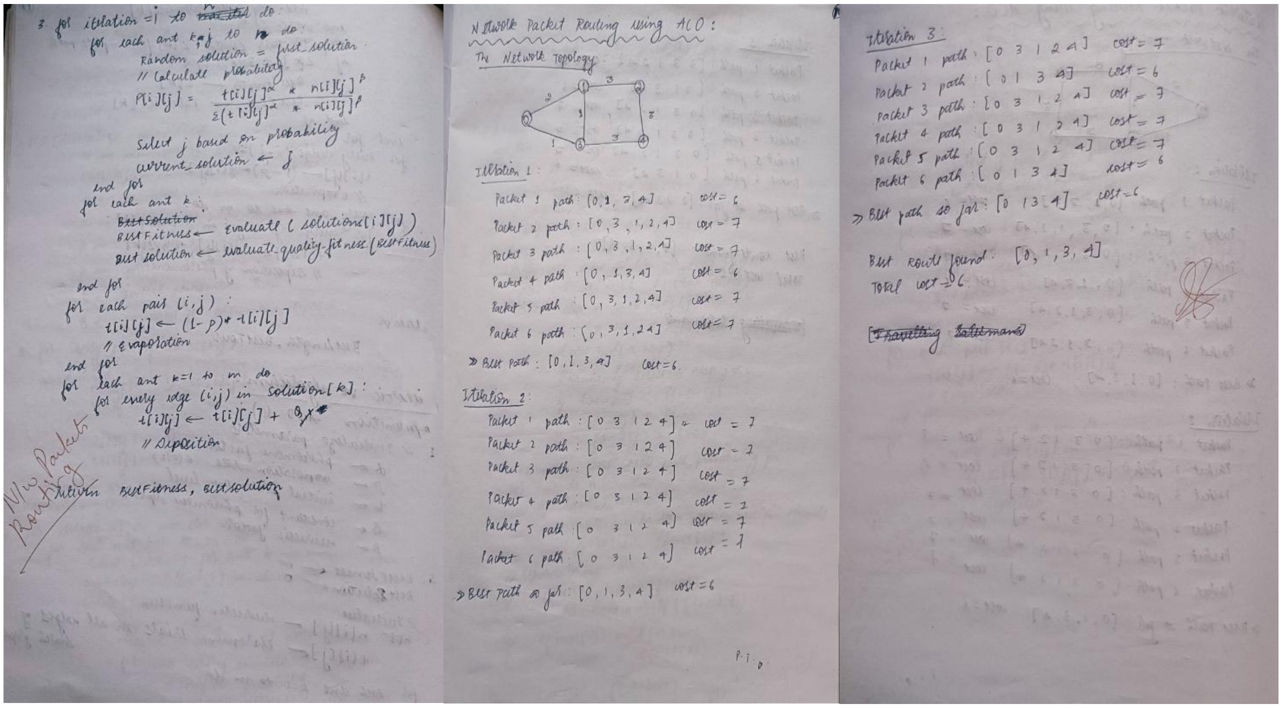
Algorithm

```
Function ACO ( ):
1. // Initialize parameters
        N ← number of cities
        dist[i][j] ← distance matrix b/w cities
        m ← number of ants
        α ← influence of pheromone
        β ← influence of heuristic (1/distance)
        ρ ← pheromone evaporation rate ∈ (0,1)
        t₀ ← Initial pheromone level
        Q ← constant for pheromone deposit
2. for every pair of cities (i,j):
      if i ≠ j :
          n[i][j] ← 1/dist [i][j]
          t[i][j] ← t₀
      else :
          n[i][j] ← 0
          t[i][j] ← 0
   BestLength ← INT_MAX
   BestTour ← 0
3. for iteration =1 to max_iter do:
      for each ant k=1 to m do :
          select start city for ant k
          Initialize Tabulist [k] = start_city
          while (TabuList [k] )
              i ← current city
              allowed_cities ← set of cities not in TabuList[k]
              for each city j in allowed_cities
                  // compute probability
                  // $P[i][j] = \frac{n[i][j]^{\alpha} * t[i][j]^{\beta}}{\Sigma(n[i][j]^\alpha * t[i][j]^\beta)}$
                  Add city j to TabuList[k]
          end while
          compute tour Length L[k] for complete path
          if L[k] < BestLength :
              BestLength ← L[k]
              BestTour ← TabuList [k]
          end if
      end for
      for every edge (i,j):
          t[i][j] ← (1 - ρ) * t[i][j]
          // Evaporation
      for every ant k=1 to m do :
          for every edge (i,j) in TabuList [k] :
              t[i][j] ← t[i][j] + Q/L[k]
              // Deposition of pheromone

   end for
   Return BestLength , BestTour
```

TSP

Generic ~~ACO~~ algorithm :

```
Function ACO ( ):
1. // Initialize parameters
        α ← pheromone factor
        ρ ← evaporation rate ∈ (0,1)
        t₀ ← initial pheromone level
        Q ← constant for pheromone deposit
        β ← heuristic factor
2. BestFitness ← ∞
   BestSolution ← 0
   // Initialise
   n[i][j] ← heuristic function
   t[i][j] ← Pheromone trials on all edges of search graph
   for each ant k=1 to m do :
```



```
3. for iteration =i to ~~max iter~~ n do:
      for each ant k,j to n do:
          Random solution = first_solution.
          // Calculate probability
          $P[i][j] = \frac{t[i][j]^{\alpha} * n[i][j]^{\beta}}{\Sigma(t[i][j]^{\alpha} * n[i][j]^{\beta}}$

          Select j based on probability
          current_solution ← j
      end for
   for each ant k :
        ~~BestSolution~~
        BestFitness ← Evaluate ( solutions[i][j] )
        Best solution ← evaluate_quality_fitness (BestFitness)
   end for
   for each pair (i,j) :
        t[i][j] ← (1-p)* t[i][j]
        // Evaporation
   end for
   for each ant k=1 to m do
       for every edge (i,j) in solution[k] :
             t[i][j] ← t[i][j] + Q/x*
             // Deposition

   return BestFitness, bestsolution
```

N/w Packets Routing

Network Packet Routing using ACO :

The Network Topology

Iteration 1 :

Packet 1 path : [0, 1, 3, 4] cost = 6
Packet 2 path : [0, 3, 1, 2, 4] cost = 7
Packet 3 path : [0, 3, 1, 2, 4] cost = 7
Packet 4 path : [0, 1, 3, 4] cost = 6
Packet 5 path : [0, 3, 1, 2, 4] cost = 7
Packet 6 path : [0, 3, 1, 2 4] cost = 7

» Best path : [0, 1, 3, 4] cost = 6.

Iteration 2 :

Packet 1 path : [0 3 1 2 4] cost = 7
Packet 2 path : [0 3 1 2 4] cost = 7
Packet 3 path : [0 3 1 2 4] cost = 7
Packet 4 path : [0 3 1 2 4] cost = 7
Packet 5 path : [0 3 1 2 4] cost = 7
Packet 6 path [0 3 1 2 4] cost = 7

» Best path so far : [0, 1, 3, 4] cost = 6

P.T.O

Iteration 3 :

Packet 1 path : [0 3 1 2 4] cost = 7
Packet 2 path : [0 1 3 4] cost = 6
Packet 3 path : [0 3 1 2 4] cost = 7
Packet 4 path : [0 3 1 2 4] cost = 7
Packet 5 path : [0 3 1 2 4] cost = 7
Packet 6 path : [0 1 3 4] cost = 6

» Best path so far : [0 1 3 4] cost = 6

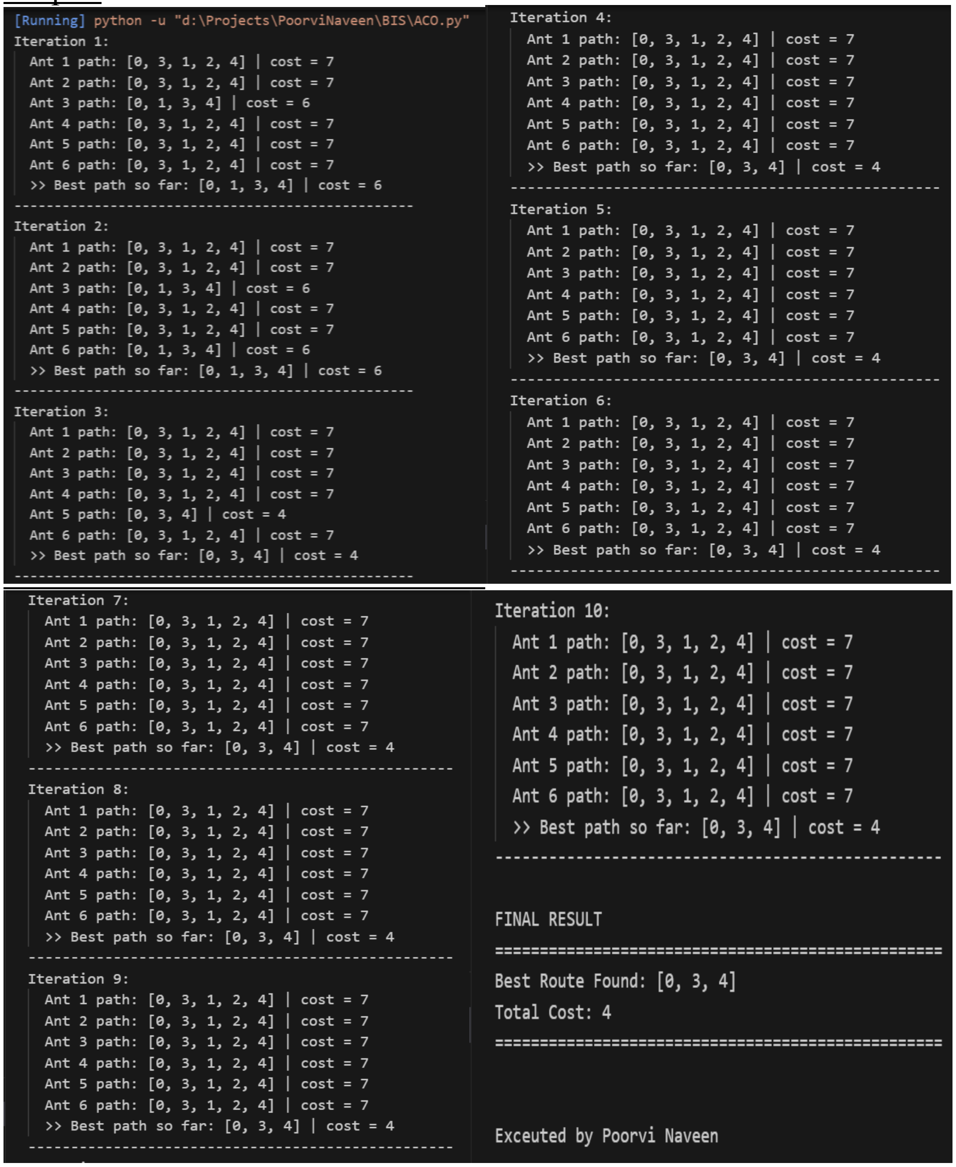
Best Route found : [0, 1, 3, 4]
Total cost = 6

[~~Travelling Salesman~~]

Code:

```
import random
import math

NUM_ANTS = 6
NUM_ITERATIONS = 10
ALPHA = 1
BETA =
RHO = 0.5
Q = 100
graph = [
    [0, 2, 0, 1, 0],
    [2, 0, 3, 1, 0],
    [0, 3, 0, 0, 2],
    [1, 1, 0, 0, 3],
    [0, 0, 4, 3, 0]
]

num_nodes = len(graph)
pheromone = [[1 for _ in range(num_nodes)] for _ in range(num_nodes)]
source = 0
destination = 4

def heuristic(i, j):
    """Heuristic value: inverse of cost"""
    if graph[i][j] == 0:
        return 0
    return 1 / graph[i][j]
def select_next_node(current, visited):
    """Select next node based on pheromone and heuristic"""
    probabilities = []
    total = 0
    for j in range(num_nodes):
        if graph[current][j] != 0 and j not in visited:
            tau = pheromone[current][j] ** ALPHA
            eta = heuristic(current, j) ** BETA
            total += tau * eta
            probabilities.append((j, tau * eta))
    if not probabilities:
        return None
    r = random.random()
    cumulative = 0
    for node, prob in probabilities:
        cumulative += prob / total
        if r <= cumulative:
            return node
    return probabilities[-1][0]
def route_cost(path):
    """Compute total cost of a given route"""
    cost = 0
    for i in range(len(path) - 1):
        cost += graph[path[i]][path[i + 1]]
    return cost
best_path = None
best_cost = math.inf
for iteration in range(NUM_ITERATIONS):
    all_paths = []
    all_costs = []
    for ant in range(NUM_ANTS):
        visited = [source]
        current = source
        while current != destination:
            next_node = select_next_node(current, visited)
            if next_node is None:
                break
            visited.append(next_node)
            current = next_node
        if visited[-1] == destination:
            cost = route_cost(visited)
            all_paths.append(visited)
            all_costs.append(cost)
    if all_costs:
        min_cost = min(all_costs)
        min_index = all_costs.index(min_cost)
        if min_cost < best_cost:
            best_cost = min_cost
            best_path = all_paths[min_index]
    for i in range(num_nodes):
        for j in range(num_nodes):
            pheromone[i][j] *= (1 - RHO)
    for path, cost in zip(all_paths, all_costs):
        for i in range(len(path) - 1):
            a, b = path[i], path[i + 1]
            pheromone[a][b] += Q / cost
            pheromone[b][a] = pheromone[a][b]      print(f"Iteration {iteration + 1}:")
    if all_costs:
        for idx, path in enumerate(all_paths):
            print(f"  Ant {idx + 1} path: {path} | cost = {all_costs[idx]}")
        print(f"  >> Best path so far: {best_path} | cost = {best_cost}")
    else:
        print("  No valid paths found in this iteration.")
    print("-" * 50)
print("\nFINAL RESULT")
print("=" * 50)
print(f"Best Route Found: {best_path}")
print(f"Total Cost: {best_cost}")
print("=" * 50)
print(“\n\nExceuted by Poorvi Naveen”)
```

Output:



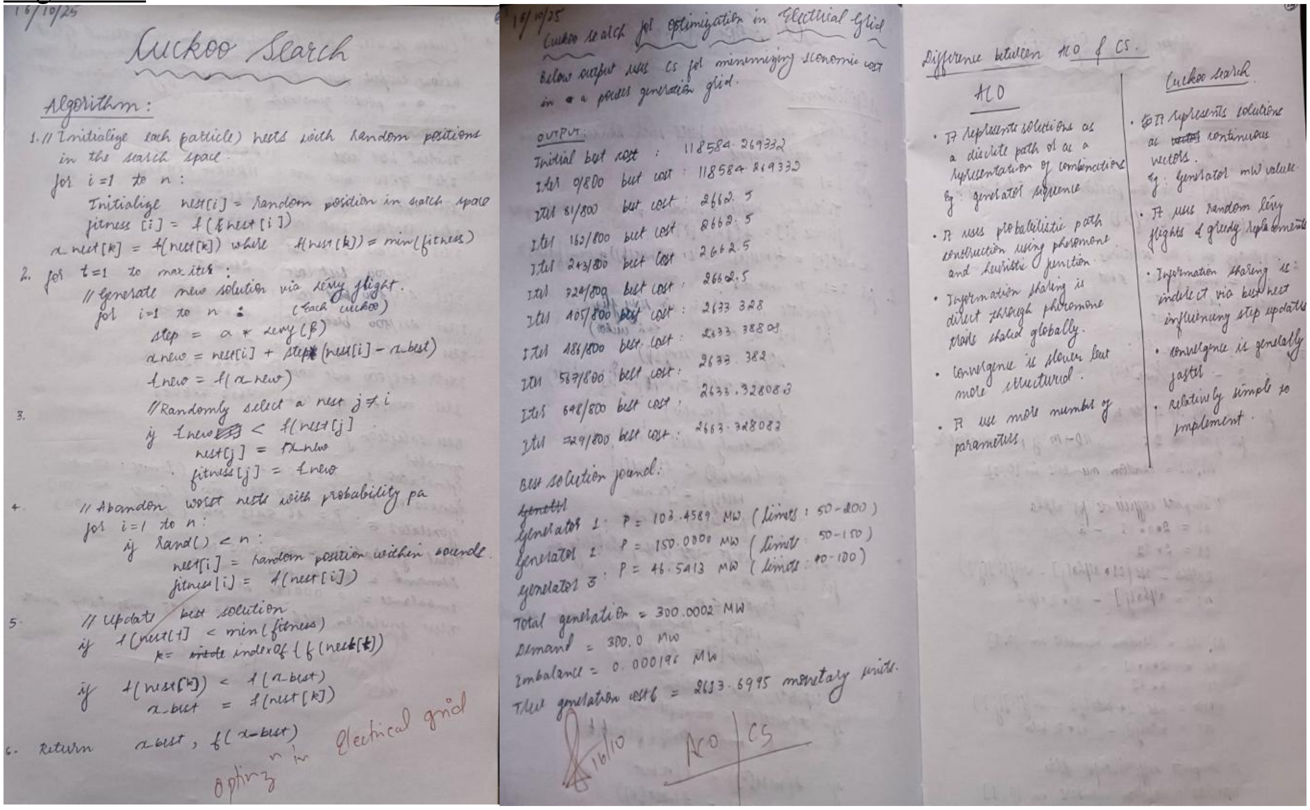
```
[Running] python -u "d:\Projects\PoorviNaveen\BIS\ACO.py"
Iteration 1:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 1, 3, 4] | cost = 6
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 1, 3, 4] | cost = 6
--------------------------------------------------
Iteration 2:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 1, 3, 4] | cost = 6
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 1, 3, 4] | cost = 6
  >> Best path so far: [0, 1, 3, 4] | cost = 6
--------------------------------------------------
Iteration 3:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 4] | cost = 4
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 4:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 5:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 6:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 7:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 8:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 9:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------
Iteration 10:
  Ant 1 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 2 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 3 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 4 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 5 path: [0, 3, 1, 2, 4] | cost = 7
  Ant 6 path: [0, 3, 1, 2, 4] | cost = 7
  >> Best path so far: [0, 3, 4] | cost = 4
--------------------------------------------------

FINAL RESULT
==================================================
Best Route Found: [0, 3, 4]
Total Cost: 4
==================================================


Exceuted by Poorvi Naveen
```

Program 4

Problem statement

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:



16/10/25

Cuckoo Search

Algorithm:

```
1. // Initialize each particle) nests with random positions
   in the search space
   for i = 1 to n :
       Initialize nest[i] = random position in search space
       fitness [i] = f(nest[i])
   x_nest[k] = f(nest[k]) where f(nest(k)) = min(fitness)
2. for t=1 to max_iter :
       // Generate new solution via levy flight.
       for i=1 to n :          (each cuckoo)
           step = α * levy(β)
           x_new = nest[i] + step* (nest[i] - x_best)
           f_new = f(x_new)
3.         // Randomly select a nest j ≠ i
           if f_new < f(nest[j]
               nest[j] = x_new
               fitness[j] = f_new
4.     // Abandon worst nests with probability pa
       for i=1 to n :
           if rand() < n :
               nest[i] = random position within bounds.
               fitness[i] = f(nest[i])
5.     // Update best solution
       if f(nest[t] < min(fitness)
           k= ~~arde~~ indexOf (f(nest[t])
       if f(nest[k]) < f(x_best)
           x_best = f(nest[k])
6. Return x_best , f(x_best)
```

optimizin in Electrical grid

16/10/25

Cuckoo search for optimization in Electrical Grid

Below output uses CS for minimizing economic cost in a power generation grid.

```
OUTPUT:
Initial best cost :  118584.269332
Iter 0/800 best cost : 118584.269332
Iter 81/800 best cost : 2662.5
Iter 162/800 best cost : 2662.5
Iter 243/800 best cost : 2662.5
Iter 324/800 best cost : 2662.5
Iter 405/800 best cost : 2633.328
Iter 486/800 best cost : 2633.38803
Iter 567/800 best cost : 2633.382
Iter 648/800 best cost : 2633.328083
Iter 729/800 best cost : 2663.328082

Best solution found:
~~generat~~
Generator 1 : P = 103.4589 MW (limits : 50-200)
Generator 2 : P = 150.0000 MW (limits 50-150)
Generator 3 : P = 46.5413 MW (limits 40-100)
Total generation = 300.0002 MW
Demand = 300.0 MW
Imbalance = 0.000196 MW
Total generation cost = 2613.6995 monetary units.
```

16/10

ACO | CS

Difference between ACO & CS.

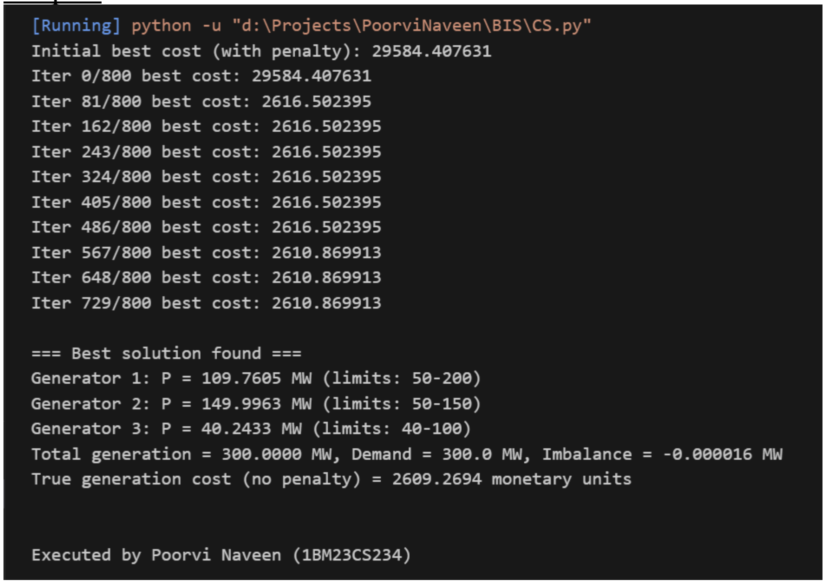
ACO	Cuckoo search
• It represents solutions as a discrete path or as a representation of combinations eg: generator sequence	• It represents solutions as ~~vector~~ continuous vectors. eg: generator mW values
• It uses probabilistic path construction using pheromone and heuristic function	• It uses random levy flights & greedy replacements
• Information sharing is direct through pheromone trails shared globally.	• Information sharing is indirect via best nest influencing step updates
• Convergence is slower but more structured.	• Convergence is generally faster
• It use more number of parameters	• Relatively simple to implement.

Code:

```
import numpy as np
import math
import random
from dataclasses import dataclass
class Generator:
    Pmin: float
    Pmax: float
    a: float
    b: float
    c: float
```

```
def total_cost(Pg, gens: list[Generator]):
    cost = 0.0
    for p, g in zip(Pg, gens):
        cost += g.a * p * p + g.b * p + g.c
    return cost
def levy_flight(beta=1.5):
    sigma_u = (math.gamma(1 + beta) * math.sin(math.pi * beta / 2) /
               (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2))) ** (1 / beta)
    u = np.random.normal(0, sigma_u)
    v = np.random.normal(0, 1)
    step = u / (abs(v) ** (1 / beta))
    return step
def simple_bounds(Pg, gens):
    Pg_bounded = np.copy(Pg)
    for i, g in enumerate(gens):
        Pg_bounded[i] = np.clip(Pg_bounded[i], g.Pmin, g.Pmax)
    return Pg_bounded
def fitness(Pg, gens, demand, penalty_factor=1e5):
    cost = total_cost(Pg, gens)
    imbalance = abs(np.sum(Pg) - demand)
    return cost + penalty_factor * imbalance
def cuckoo_search(gens, demand, n_nests=25, max_iter=500, pa=0.25, beta=1.5, verbose=False):
    dim = len(gens)
    nests = np.zeros((n_nests, dim))
    for i in range(n_nests):
        for j, g in enumerate(gens):
            nests[i, j] = np.random.uniform(g.Pmin, g.Pmax)
    fitnesses = np.array([fitness(nests[i], gens, demand) for i in range(n_nests)])
    best_idx = np.argmin(fitnesses)
    best_nest = nests[best_idx].copy()
    best_fit = fitnesses[best_idx]
    if verbose:
        print(f"Initial best cost (with penalty): {best_fit:.6f}")
    for it in range(max_iter):
        for i in range(n_nests):
            step = levy_flight(beta)
            step_size = 0.01 * step * (nests[i] - best_nest)
            new_nest = nests[i] + step_size * np.random.randn(dim)
            new_nest = simple_bounds(new_nest, gens)
            new_fit = fitness(new_nest, gens, demand)
            if new_fit < fitnesses[i]:
                nests[i] = new_nest
                fitnesses[i] = new_fit
                if new_fit < best_fit:
                    best_fit = new_fit
                    best_nest = new_nest.copy()
        K = np.random.rand(n_nests) < pa
        for i in range(n_nests):
            if K[i]:
                idx1, idx2 = np.random.choice(n_nests, 2, replace=False)
                step = np.random.rand(dim) * (nests[idx1] - nests[idx2])
                new_nest = nests[i] + step
                new_nest = simple_bounds(new_nest, gens)
                new_fit = fitness(new_nest, gens, demand)
                if new_fit < fitnesses[i]:
                    nests[i] = new_nest
                    fitnesses[i] = new_fit
                    if new_fit < best_fit:
                        best_fit = new_fit
                        best_nest = new_nest.copy()
        if verbose and (it % (max_iter//10 + 1) == 0):
            print(f"Iter {it}/{max_iter} best cost: {best_fit:.6f}")
    imbalance = np.sum(best_nest) - demand
    true_cost = total_cost(best_nest, gens)
    return {
        "Pg": best_nest,
        "cost_with_penalty": best_fit,
        "true_cost": true_cost,
        "imbalance": imbalance,
        "fitnesses": fitnesses
    }
gens = [
    Generator(Pmin=50, Pmax=200, a=0.002, b=8.0, c=100.0),
    Generator(Pmin=50, Pmax=150, a=0.003, b=6.5, c=120.0),
    Generator(Pmin=40, Pmax=100, a=0.0015, b=9.0, c=80.0),
]
demand = 300.0  # MW
result = cuckoo_search(gens, demand, n_nests=40, max_iter=800, pa=0.25, beta=1.5, verbose=True)

print("\n=== Best solution found ===")
for i, p in enumerate(result["Pg"], start=1):
    print(f"Generator {i}: P = {p:.4f} MW (limits: {gens[i-1].Pmin}-{gens[i-1].Pmax})")
print(f"Total generation = {np.sum(result['Pg']):.4f} MW, Demand = {demand} MW, Imbalance = {result['imbalance']:.6f} MW")
print(f"True generation cost (no penalty) = {result['true_cost']:.4f} monetary units")
print("\n\nExecuted by Poorvi Naveen (1BM23CS234)")
```

Output:



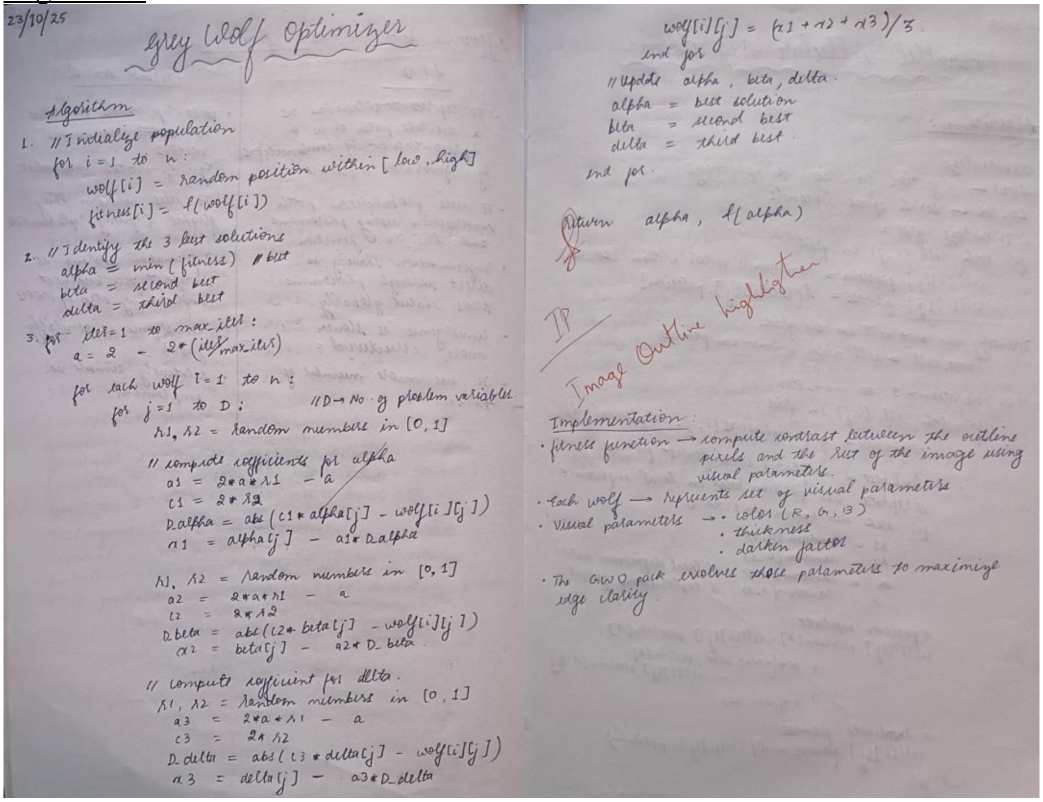
```
[Running] python -u "d:\Projects\PoorviNaveen\BIS\CS.py"
Initial best cost (with penalty): 29584.407631
Iter 0/800 best cost: 29584.407631
Iter 81/800 best cost: 2616.502395
Iter 162/800 best cost: 2616.502395
Iter 243/800 best cost: 2616.502395
Iter 324/800 best cost: 2616.502395
Iter 405/800 best cost: 2616.502395
Iter 486/800 best cost: 2616.502395
Iter 567/800 best cost: 2610.869913
Iter 648/800 best cost: 2610.869913
Iter 729/800 best cost: 2610.869913

=== Best solution found ===
Generator 1: P = 109.7605 MW (limits: 50-200)
Generator 2: P = 149.9963 MW (limits: 50-150)
Generator 3: P = 40.2433 MW (limits: 40-100)
Total generation = 300.0000 MW, Demand = 300.0 MW, Imbalance = -0.000016 MW
True generation cost (no penalty) = 2609.2694 monetary units


Executed by Poorvi Naveen (1BM23CS234)
```

Program 5

Problem statement

Grey Wolf Optimizer (GWO):
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behaviour 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:



23/10/25

Grey Wolf Optimizer

Algorithm

```
1. // Initialize population
   for i = 1 to n:
       wolf[i] = random position within [low, high]
       fitness[i] = f(wolf[i])
2. // Identify the 3 best solutions
   alpha = min(fitness)  // best
   beta  = second best
   delta = third best
3. for iter = 1 to max_iter:
     a = 2 - 2*(iter/max_iter)

     for each wolf i = 1 to n:
         for j = 1 to D:          // D → No. of problem variables
             r1, r2 = random numbers in [0, 1]

             // compute coefficients for alpha
             a1 = 2*a*r1 - a
             c1 = 2*r2
             D_alpha = abs(c1*alpha[j] - wolf[i][j])
             x1 = alpha[j] - a1*D_alpha

             r1, r2 = random numbers in [0, 1]
             a2 = 2*a*r1 - a
             c2 = 2*r2
             D_beta = abs(c2*beta[j] - wolf[i][j])
             x2 = beta[j] - a2*D_beta

             // compute coefficient for delta.
             r1, r2 = random numbers in [0, 1]
             a3 = 2*a*r1 - a
             c3 = 2*r2
             D_delta = abs(c3*delta[j] - wolf[i][j])
             x3 = delta[j] - a3*D_delta

             wolf[i][j] = (x1 + x2 + x3)/3
         end for
     // Update alpha, beta, delta
     alpha = best solution
     beta  = second best
     delta = third best
   end for

   return alpha, f(alpha)
```

IP

Image Outline highlighter

Implementation:

- fitness function → compute contrast between the outline pixels and the rest of the image using visual parameters.
- Each wolf → represents set of visual parameters
- Visual parameters →
  - color (R, G, B)
  - thickness
  - darken factor
- The GWO pack evolves those parameters to maximize edge clarity.

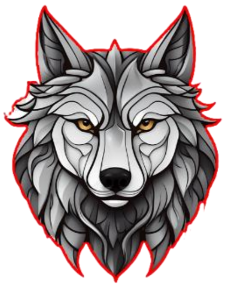
Code:

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow

def highlight_object_outline(image_path, outline_color=(0, 0, 255), thickness=5, darken_factor=0.5)
    img = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
    if img is None:
        raise FileNotFoundError(f"Image not found at {image_path}")
    if img.shape[2] < 4:
        raise ValueError("Image must have an alpha channel for transparency.")
    b, g, r, a = cv2.split(img)
    mask = cv2.threshold(a, 1, 255, cv2.THRESH_BINARY)[1]
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    rgb = cv2.merge((b, g, r))
    darkened = (rgb * darken_factor).astype(np.uint8)
    outlined_rgb = rgb.copy()
    cv2.drawContours(outlined_rgb, contours, -1, outline_color, thickness)
    edge_mask = np.zeros_like(mask)
    cv2.drawContours(edge_mask, contours, -1, 255, thickness)
    color_array = np.full_like(outlined_rgb[edge_mask > 0], outline_color, dtype=np.uint8)
    outlined_rgb[edge_mask > 0] = cv2.addWeighted(
        outlined_rgb[edge_mask > 0], 0.5,
        color_array, 0.5, 0
    )
    outlined_img = cv2.merge((outlined_rgb, a))
    return outlined_img
outlined = highlight_object_outline(
    "1.png", outline_color=(0, 0, 255), thickness=10, darken_factor=0.7
)
cv2_imshow(outlined)
```

Output:

Input.png

download.png

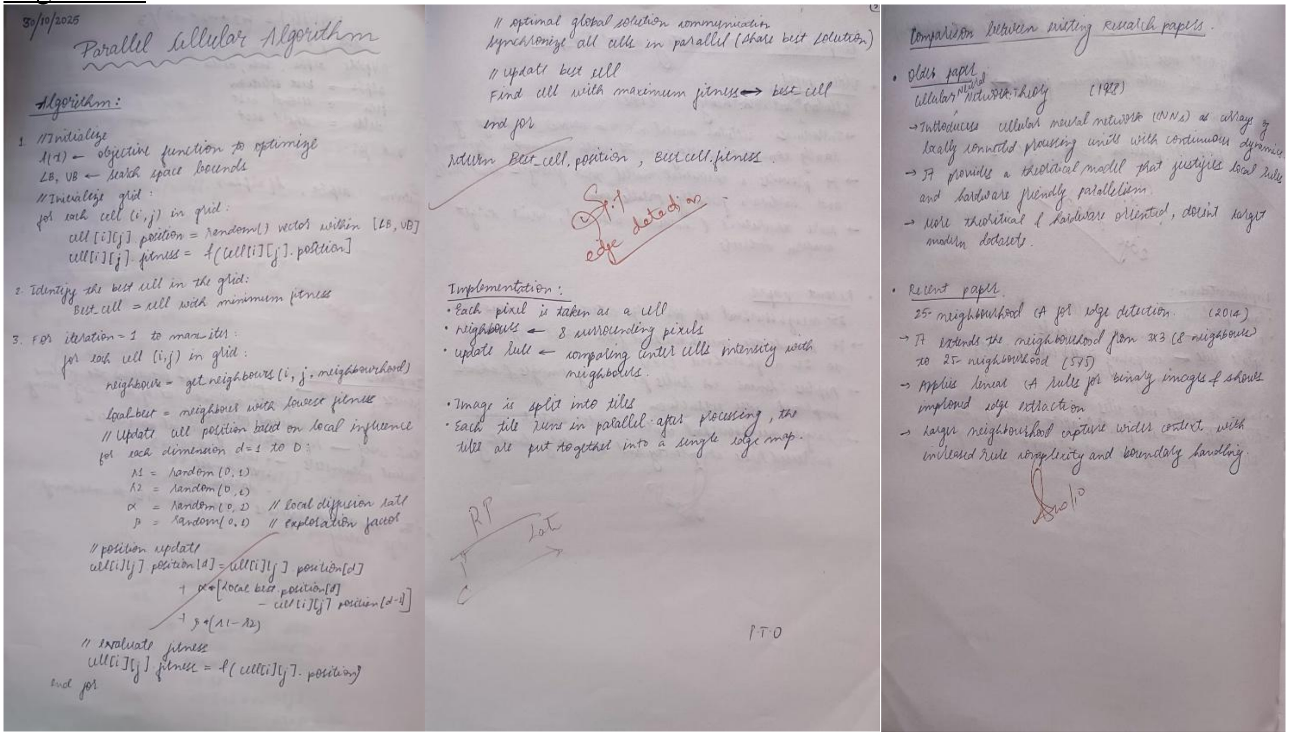


Program 6

Problem statement

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 neighbours 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:



30/10/2025

Parallel Cellular Algorithm

Algorithm:

```
1. //Initialize
   f(x) ← objective function to optimize
   LB, UB ← search space bounds
   //Initialize grid:
   for each cell (i,j) in grid:
       cell[i][j].position = random() vector within [LB, UB]
       cell[i][j].fitness = f(cell[i][j].position]

2. Identify the best cell in the grid:
       Best_cell = cell with minimum fitness

3. For iteration = 1 to max_iter:
       for each cell (i,j) in grid:
           neighbours = get_neighbours(i, j, neighbourhood)
           local_best = neighbours with lowest fitness
           // Update cell position based on local influence
           for each dimension d=1 to D:
               r1 = random(0, 1)
               r2 = random(0, 1)
               α = random(0, 1)   // local diffusion rate
               β = random(0, 1)   // exploration factor
           // position update
           cell[i][j].position[d] = cell[i][j].position[d]
                      + α*[local best.position[d]
                               - cell[i][j].position[d-1]]
                      + β*(r1 - r2)
           // evaluate fitness
           cell[i][j].fitness = f(cell[i][j].position)
   end for

   // optimal global solution communication
   synchronize all cells in parallel (share best solution)
   // update best cell
   Find cell with maximum fitness → best_cell
   end for

return Best_cell.position , Best_cell.fitness
```

edge detection

Implementation:

- Each pixel is taken as a cell
- neighbours ← 8 surrounding pixels
- update rule ← comparing center cells intensity with neighbours.
- Image is split into tiles
- Each tile runs in parallel. after processing, the tiles are put together into a single edge map.

RP

Lot

PTO

Comparison between existing Research papers.

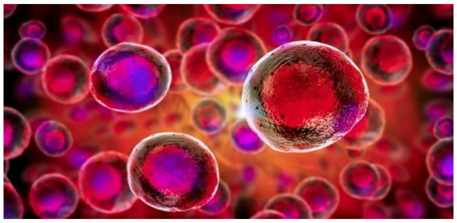
- Older paper
  Cellular Neural Network: Theory (1988)
  → Introduces cellular neural networks (CNNs) as arrays of locally connected processing units with continuous dynamics.
  → It provides a theoretical model that justifies local rules and hardware friendly parallelism.
  → More theoretical & hardware oriented, doesn't target modern datasets.
- Recent paper
  25-neighbourhood CA for edge detection. (2014)
  → It extends the neighbourhood from 3x3 (8 neighbours) to 25-neighbourhood (5x5)
  → Applies linear CA rules for binary images & shows improved edge extraction
  → Larger neighbourhood capture wider context with increased rule complexity and boundary handling.

Code:

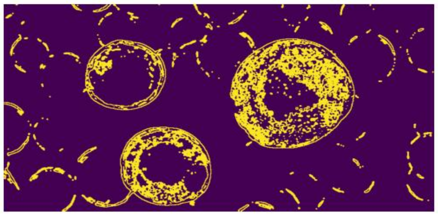
```
import numpy as np
import cv2
import multiprocessing as mp
from functools import partial
import os
from typing import Tuple
import matplotlib.pyplot as plt
def pad_image(img: np.ndarray, pad: int) -> np.ndarray:
    return np.pad(img, pad_width=pad, mode='reflect')
def ca_step(tile: np.ndarray, threshold: int) -> np.ndarray:
    center = tile
    maxdiff = np.zeros_like(tile, dtype=np.int16)
    shifts = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]
    for dy, dx in shifts:
        neigh = np.roll(np.roll(tile, dy, axis=0), dx, axis=1)
        diff = np.abs(center.astype(np.int16) - neigh.astype(np.int16))
        maxdiff = np.maximum(maxdiff, diff)
    out = (maxdiff > threshold).astype(np.uint8) * 255
    return out
def run_ca_on_tile(tile_with_meta: Tuple[np.ndarray, int, int, int, int, int], iterations: int, threshold: int) ->
Tuple[int,int,np.ndarray]:
    padded_tile, tile_y, tile_x, y0, x0, overlap = tile_with_meta
    tile = padded_tile.copy()
    for i in range(iterations):
        tile = ca_step(tile, threshold)
    if overlap > 0:
        core = tile[overlap:-overlap, overlap:-overlap].copy()
    else:
        core = tile
    return (tile_y, tile_x, core)
def split_image_to_tiles(img: np.ndarray, tile_size: int, overlap: int):
    h, w = img.shape
    tiles = []
    rows = list(range(0, h, tile_size))
    cols = list(range(0, w, tile_size))
    pad = overlap
    padded_img = pad_image(img, pad)
    for i, y in enumerate(rows):
        for j, x in enumerate(cols):
            y0, x0 = y, x
            y1 = y0 + tile_size
            x1 = x0 + tile_size
            py0 = y0
            px0 = x0
            sy = py0
            sx = px0
            ey = py0 + tile_size + 2*pad
            ex = px0 + tile_size + 2*pad
            sy = max(0, sy)
            sx = max(0, sx)
            ey = min(padded_img.shape[0], ey)
            ex = min(padded_img.shape[1], ex)
            tile = padded_img[sy:ey, sx:ex].copy()
            expected_h = tile_size + 2*pad
            expected_w = tile_size + 2*pad
            if tile.shape[0] != expected_h or tile.shape[1] != expected_w:
                tile = pad_image(tile, 0)
                tile = cv2.copyMakeBorder(tile, 0, expected_h - tile.shape[0],
                                          0, expected_w - tile.shape[1],
                                          borderType=cv2.BORDER_REFLECT)
            tiles.append((tile, i, j, y0, x0, pad))
    return tiles, rows, cols
def stitch_tiles(tiles_out, img_shape: Tuple[int,int], tile_size: int, overlap: int, rows, cols):
    h, w = img_shape
    out = np.zeros((h, w), dtype=np.uint8)
    for tile_y, tile_x, core in tiles_out:
        y = rows[tile_y]
        x = cols[tile_x]
        y1 = min(h, y + tile_size)
        x1 = min(w, x + tile_size)
        core_h = y1 - y
        core_w = x1 - x
        out[y:y1, x:x1] = core[:core_h, :core_w]
    return out
def parallel_pca_edge_detect(img_gray: np.ndarray, tile_size: int = 128,
                             overlap: int = 8, iterations: int = 2,
                             threshold: int = 20, processes: int = None) -> np.ndarray:
    if processes is None:
        processes = max(1, mp.cpu_count() - 1)
    tiles, rows, cols = split_image_to_tiles(img_gray, tile_size, overlap)
    print(f"[PCA] Running on {len(tiles)} tiles with {processes} processes...")
    tile_args = tiles
    with mp.Pool(processes=processes) as pool:
        fn = partial(run_ca_on_tile, iterations=iterations, threshold=threshold)
        results = pool.map(fn, tile_args)
    out = stitch_tiles(results, img_gray.shape, tile_size, overlap, rows, cols)
    return out
def main(input_path: str, output_path: str,
         tile_size: int=128, overlap: int=8, iterations: int=2, threshold: int=20):
    if not os.path.isfile(input_path):
        raise FileNotFoundError(f"Input not found: {input_path}")
    img = cv2.imread(input_path, cv2.IMREAD_COLOR)
    if img is None:
        raise ValueError("Could not read input image (cv2 returned None)")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    edges = parallel_pca_edge_detect(gray, tile_size=tile_size, overlap=overlap,
                                     iterations=iterations, threshold=threshold)
    edges = cv2.medianBlur(edges, 3)
    cv2.imwrite(output_path, edges)
    print(f"Saved edge image to {output_path}")
input_path = "/1.jpg"
output_path = "/O1.jpg"
tile_size = 128
overlap = 8
iterations = 2
threshold = 20
main(input_path, output_path, tile_size, overlap, iterations, threshold)
main(input_path, output_path, tile_size, overlap, iterations, threshold)
img = cv2.imread(output_path, cv2.IMREAD_GRAYSCALE)
plt.imshow(img)
plt.axis('off')
plt.show()
print("\n\nExecuted by Poorvi Naveen")
```

Output:

Input.png



processed.png

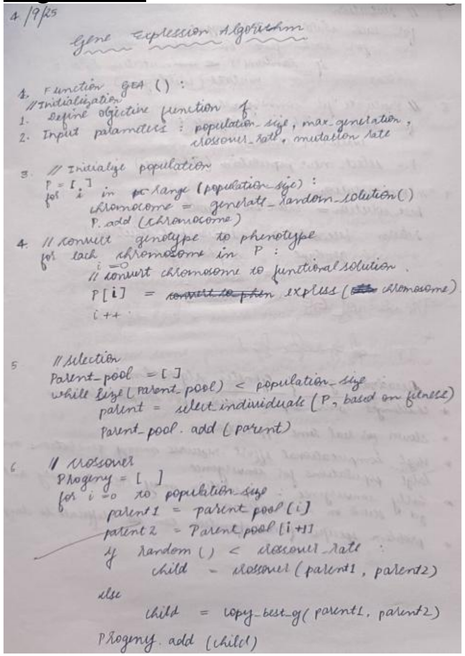


Program 7

Problem statement

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.

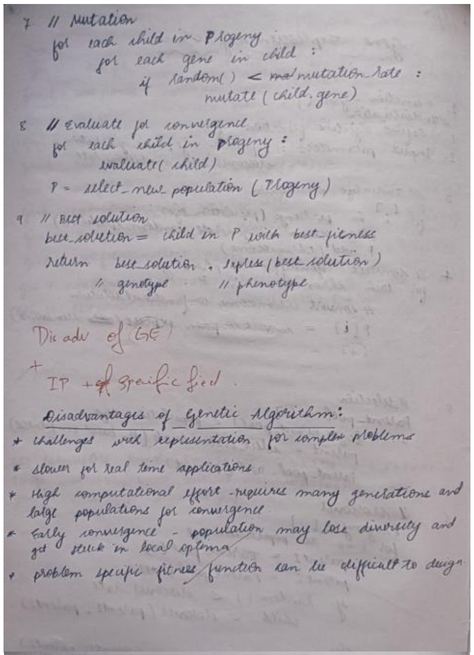
Algorithm:



4/9/25

Gene Expression Algorithm

```
1. Function GEA ():
   //Initialization
1.   Define objective function f
2.   Input parameters : population_size, max_generation,
                        crossover_rate, mutation_rate

3.   // Initialize population
     P = [ ]
     for i in ~~pc~~ range (population_size) :
         chromosome = generate_random_solution()
         P.add (chromosome)
4.   // convert genotype to phenotype
     for each chromosome in P :
         i = 0
         // convert chromosome to functional solution.
         P[i] = ~~convert to phen~~ express (~~[illegible]~~ chromosome)
         i ++

5.   // selection
     Parent_pool = [ ]
     while size (Parent_pool) < population_size
         parent = select individuals (P, based on fitness)
         Parent_pool. add (parent)

6.   // crossover
     Progeny = [ ]
     for i = 0 to population_size
         parent1 = parent pool [i]
         parent2 = Parent pool [i+1]
         if random ( ) < crossover_rate :
             child = crossover (parent1, parent2)
         else
             child = copy_best_of ( parent1, parent2)
         Progeny. add (child)
```



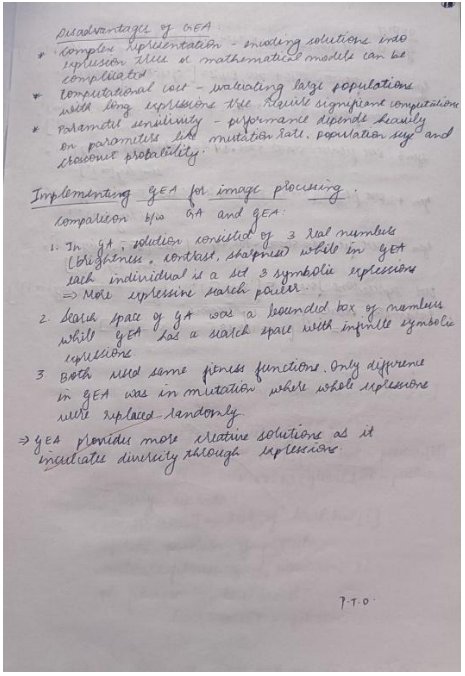
```
7.   // Mutation
     for each child in Progeny
         for each gene in child :
             if random( ) < ~~ma~~ mutation_rate :
                 mutate (child. gene)

8.   // Evaluate for convergence
     for each child in progeny :
         evaluate( child)
     P = select_new population ( Progeny)

9.   // Best solution
     best_solution = child in P with best_fitness
     return best_solution , express (best_solution)
          // genotype        // phenotype
```

Dis adv of GE

+ IP + ~~of~~ specific field.

Disadvantages of Genetic Algorithm:

- challenges with representation for complex problems
- slower for real time applications
- High computational effort - requires many generations and large populations for convergence
- Early convergence - population may lose diversity and get stuck in local optima
- problem specific fitness function can be difficult to design



Disadvantages of GEA

- complex representation - encoding solutions into expression trees or mathematical models can be complicated
- computational cost - evaluating large populations with long expressions tree require significant computations
- Parameter sensitivity - performance depends heavily on parameters like mutation rate, population size and crossover probability.

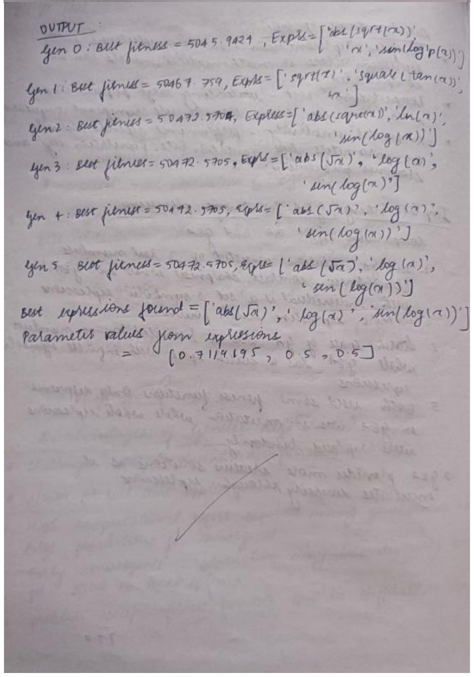
Implementing GEA for image processing.

comparison b/w GA and GEA:

1. In GA, solution consisted of 3 real numbers (brightness, contrast, sharpness) while in GEA each individual is a set 3 symbolic expressions
   ⇒ More expressive search power.
2. Search space of GA was a bounded box of numbers while GEA has a search space with infinite symbolic expressions
3. Both used same fitness functions. Only difference in GEA was in mutation where whole expressions were replaced randomly

⇒ GEA provides more creative solutions as it inculcates diversity through expressions.

P.T.O.



OUTPUT :

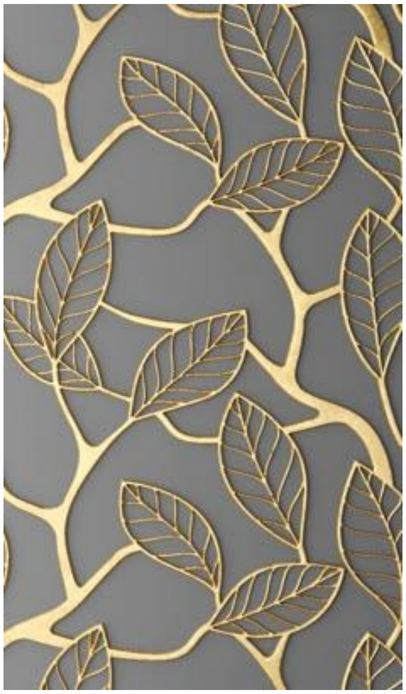
```
Gen 0 : Best fitness = 5045.9424 , Exprs = ['abs (sqrt(x))',
                                           'x', 'sin(log1p(x))']
Gen 1 : Best fitness = 50467.759, Exprs = ['sqrt(x)', 'square (tan(x))',
                                          'x']
Gen 2 : Best fitness = 50472.5704, Exprs = ['abs (sqrt(x))', 'ln(x)',
                                          'sin(log(x))']
Gen 3 : Best fitness = 50472.5705, Exprs = ['abs (√x)', 'log (x)',
                                          'sin( log(x))']
Gen 4 : Best fitness = 50472.5705, Exprs = ['abs (√x)', 'log (x)',
                                          'sin( log (x))']
Gen 5 : Best fitness = 50472.5705, Exprs = ['abs (√x)', 'log (x)',
                                          'sin( log(x))']
Best expressions found = ['abs(√x)', 'log(x)', 'sin(log(x))']
Parameters values from expressions
       = [0.7114195, 0.5, 0.5]
```

Code:

```
import random
import numpy as np
from PIL import Image, ImageEnhance
from skimage.metrics import structural_similarity as ssim
from skimage.filters import sobel
from skimage import img_as_float
import math

original = Image.open("myImage.jpg").convert("RGB")
original_np = np.array(original)
try:
    target = Image.open("target.jpg").convert("RGB")
    target_np = np.array(target)
    USE_REFERENCE = True
except:
    USE_REFERENCE = False
def protected_div(x, y):
    try:
        return x / y if abs(y) > 1e-6 else 1.0
    except:
        return 1.0
FUNCTIONS = [
    (lambda x: x, 'x'),
    (np.sin, 'sin'),
    (np.cos, 'cos'),
    (np.tan, 'tan'),
    (np.exp, 'exp'),
    (np.log1p, 'log1p'),
    (lambda x: x**2, 'square'),
    (np.sqrt, 'sqrt'),
    (np.abs, 'abs'),
]
def random_expr(depth=2):
    if depth == 0 or random.random() < 0.3:
        return 'x'
    func = random.choice(FUNCTIONS)[1]
    sub = random_expr(depth - 1)
    return f"{func}({sub})"
def evaluate_expr(expr_str, x_val):
    try:
        x = x_val
        return eval(expr_str, {"x": x, "sin": np.sin, "cos": np.cos, "tan": np.tan,
                               "exp": np.exp, "log1p": np.log1p, "sqrt": np.sqrt,
                               "abs": np.abs, "square": lambda x: x**2})
    except Exception as e:
        return 1.0
def init_population(size):
    return [[random_expr(2) for _ in range(3)] for _ in range(size)]
def apply_adjustments_from_expr(exprs):
    x = np.mean(original_np) / 255.0
    b = np.clip(evaluate_expr(exprs[0], x), 0.5, 2.0)
    c = np.clip(evaluate_expr(exprs[1], x), 0.5, 2.0)
    s = np.clip(evaluate_expr(exprs[2], x), 0.5, 2.0)
    img = original.copy()
    img = ImageEnhance.Brightness(img).enhance(b)
    img = ImageEnhance.Contrast(img).enhance(c)
    img = ImageEnhance.Sharpness(img).enhance(s)
    return img, [b, c, s]
def fitness(ind):
    img, _ = apply_adjustments_from_expr(ind)
    img_np = np.array(img)
    if USE_REFERENCE:
        return ssim(target_np, img_np, channel_axis=2)
    else:
        gray = img.convert("L")
        gray_np = img_as_float(np.array(gray))
        entropy = -np.sum(gray_np * np.log2(gray_np + 1e-10))
        edge_strength = np.mean(sobel(gray_np))
        return entropy + edge_strength
def select(pop, fitnesses):
    i, j = random.sample(range(len(pop)), 2)
    return pop[i] if fitnesses[i] > fitnesses[j] else pop[j]
def crossover(p1, p2):
    child = []
    for a, b in zip(p1, p2):
        if random.random() < 0.5:
            child.append(a)
        else:
            child.append(b)
    return child
def mutate(expr):
    expr_list = expr.split()
    if random.random() < 0.3:
        return random_expr(2)
    return expr
def mutate_ind(ind):
    return [mutate(expr) if random.random() < 0.3 else expr for expr in ind]
def run_gep():
    POP_SIZE = 20
    N_GEN = 15
    ELITE = 2
    pop = init_population(POP_SIZE)
    for gen in range(N_GEN):
        fitnesses = [fitness(ind) for ind in pop]
        ranked = sorted(zip(pop, fitnesses), key=lambda x: x[1], reverse=True)
        best_ind, best_fit = ranked[0]
        print(f"Gen {gen}: Best fitness = {best_fit:.4f}, Exprs = {best_ind}")
        new_pop = [ind.copy() for ind, _ in ranked[:ELITE]]
        while len(new_pop) < POP_SIZE:
            p1, p2 = select(pop, fitnesses), select(pop, fitnesses)
            child = crossover(p1, p2)
            child = mutate_ind(child)
            new_pop.append(child)
        pop = new_pop
    best_img, best_params = apply_adjustments_from_expr(best_ind)
    best_img.save("gep_optimized.jpg")
    print("Best expressions found:", best_ind)
    print("Parameter values from expressions:", best_params)
run_gep()
print("\n\nExceuted by Poorvi Naveen")
```

Output:

myImage.jpg



gep_optimized.jpg