# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB RECORD**

# Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**R Bhuvan Aditya (1BM23CS254)**

***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 **R Bhuvan Aditya (1BM23CS254),** 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.

| | |
|---|---|
| Ms. Swathi Sridharan<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Jyothi S Nayak<br>Professor & HOD<br>Department of CSE, BMSCE |

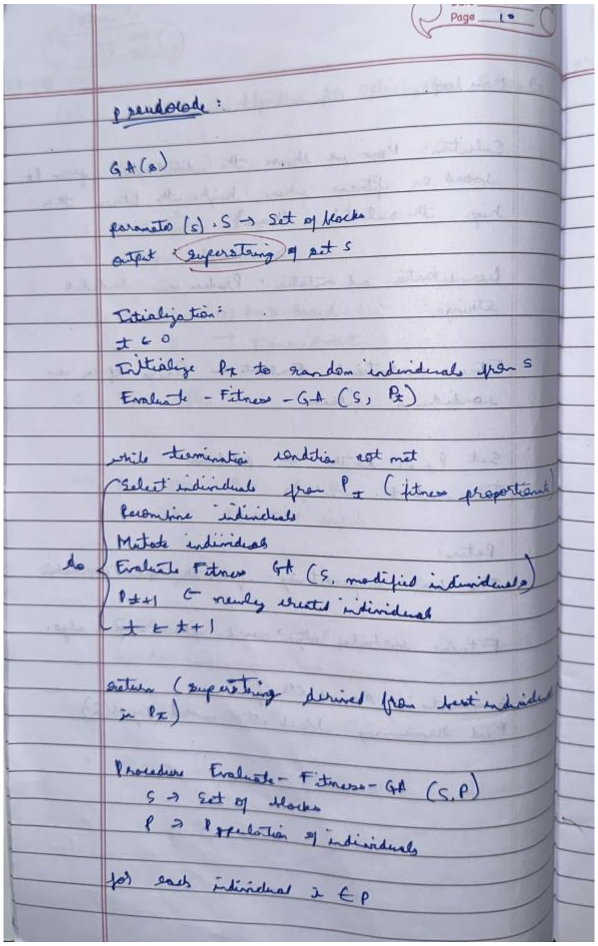
# Index



Github Link:
https://github.com/rbhuvan10/1BM23CS254_BIS

**Program 1**
Genetic Algorithm for Optimization

Algorithm:



Page 10

Pseudocode :

GA(s)

parameter (s) . S → Set of blocks
output : superstring of set S

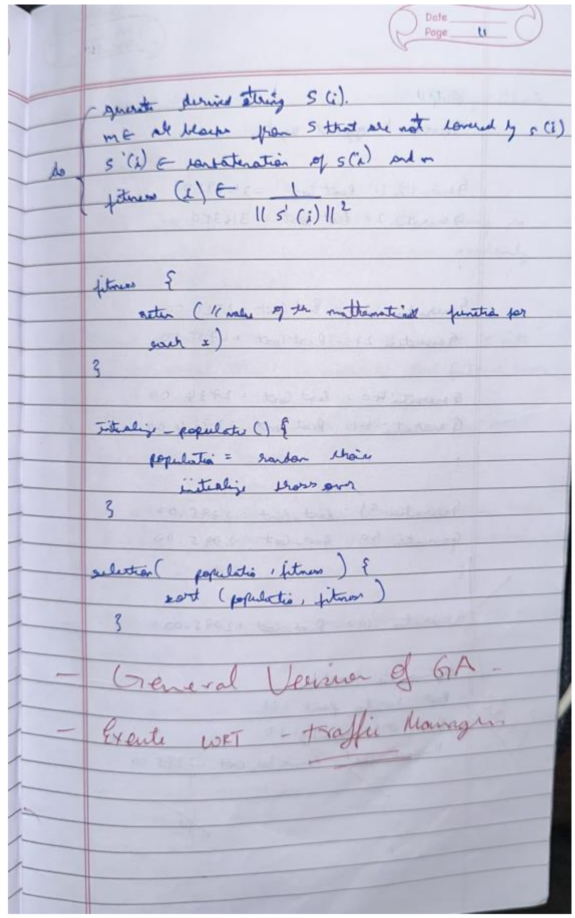
Initialization :
t ← 0
Initialize $P_t$ to random individuals from S
Evaluate - Fitness - GA (S, $P_t$)

while termination condition not met
do {
Select individuals from $P_t$ (fitness proportionate)
Recombine individuals
Mutate individuals
Evaluate Fitness GA (S, modified individuals)
$P_{t+1}$ ← newly created individuals
t ← t+1
}

return (superstring derived from best individual in $P_t$)

Procedure Evaluate - Fitness - GA (S, P)
S → Set of blocks
P → Population of individuals

for each individual i ∈ P



Page 11

do {
generate derived string S (i).
m ← all blocks from S that are not covered by s (i)
s'(i) ← concatenation of s(i) and m
fitness (i) ← $\frac{1}{\|s'(i)\|^2}$
}

fitness {
return (// value of the mathematical function for each x)
}

initialize _ population () {
population = random choice
initialize cross over
}

selection( population , fitness ) {
sort (population, fitness )
}

- General Version of GA
- Execute WRT - traffic Management

Code:
import random
import numpy as np
import matplotlib.pyplot as plt

RANDOM_SEED = 42
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

CYCLE_MIN = 30.0
CYCLE_MAX = 120.0
NS_GREEN_MIN = 5.0

POP_SIZE = 60
GENERATIONS = 50
TOURNAMENT_K = 3

```
ELITISM = 2
CROSSOVER_RATE = 0.9
MUTATION_RATE = 0.25
MUTATION_STD_CYCLE = 4.0
MUTATION_STD_NS = 3.0

TARGET_DELAY = 50.0

def compute_delay(cycle, ns_green):
    cycle = float(np.clip(cycle, CYCLE_MIN, CYCLE_MAX))
    ns_green = float(np.clip(ns_green, NS_GREEN_MIN, cycle - NS_GREEN_MIN))
    ns_ratio = ns_green / cycle
    base = TARGET_DELAY
    cycle_penalty = 0.35 * (cycle - 60)
    balance_penalty = 120 * (ns_ratio - 0.5) ** 2
    delay = base + cycle_penalty + balance_penalty
    return max(1.0, delay)

def fitness(ind):
    d = compute_delay(*ind)
    return -abs(d - TARGET_DELAY)

def random_individual():
    cycle = random.uniform(CYCLE_MIN, CYCLE_MAX)
    ns = random.uniform(NS_GREEN_MIN, cycle - NS_GREEN_MIN)
    return [cycle, ns]

def ensure_bounds(ind):
    cycle, ns = ind
    cycle = float(np.clip(cycle, CYCLE_MIN, CYCLE_MAX))
    ns = float(np.clip(ns, NS_GREEN_MIN, cycle - NS_GREEN_MIN))
    return [cycle, ns]

def tournament(pop, fits):
    choices = random.sample(range(len(pop)), TOURNAMENT_K)
    best = max(choices, key=lambda i: fits[i])
    return pop[best][:]

def crossover(p1, p2):
    if random.random() > CROSSOVER_RATE:
        return p1[:], p2[:]
    c1, c2 = p1[:], p2[:]
    if random.random() < 0.5: c1[0], c2[0] = p2[0], p1[0]
    if random.random() < 0.5: c1[1], c2[1] = p2[1], p1[1]
    return c1, c2

def mutate(ind):
    if random.random() < MUTATION_RATE:
        ind[0] += random.gauss(0, MUTATION_STD_CYCLE)
```

```python
    if random.random() < MUTATION_RATE:
        ind[1] += random.gauss(0, MUTATION_STD_NS)
    return ensure_bounds(ind)

def run_ga():
    pop = [random_individual() for _ in range(POP_SIZE)]
    best_final = None

    for gen in range(1, GENERATIONS + 1):
        fits = [fitness(ind) for ind in pop]
        sorted_idx = sorted(range(len(pop)), key=lambda i: fits[i])
        sorted_idx.reverse()
        best = pop[sorted_idx[0]]
        best_final = best
        best_delay = compute_delay(*best)
        print(f"Gen {gen} best avg delay: {best_delay:.2f}")

        new_pop = [pop[i][:] for i in sorted_idx[:ELITISM]]

        while len(new_pop) < POP_SIZE:
            p1 = tournament(pop, fits)
            p2 = tournament(pop, fits)
            c1, c2 = crossover(p1, p2)
            new_pop.append(mutate(c1))
            if len(new_pop) < POP_SIZE:
                new_pop.append(mutate(c2))

        pop = new_pop

    cycle, ns = best_final
    ew = cycle - ns
    final_delay = compute_delay(cycle, ns)

    print("\noptimal signal plan:")
    print(f"cycle length: {cycle:.2f}")
    print(f"north south green: {ns:.2f}")
    print(f"east west green: {ew:.2f}")
    print(f"average delay: {final_delay:.2f}")

run_ga()
```

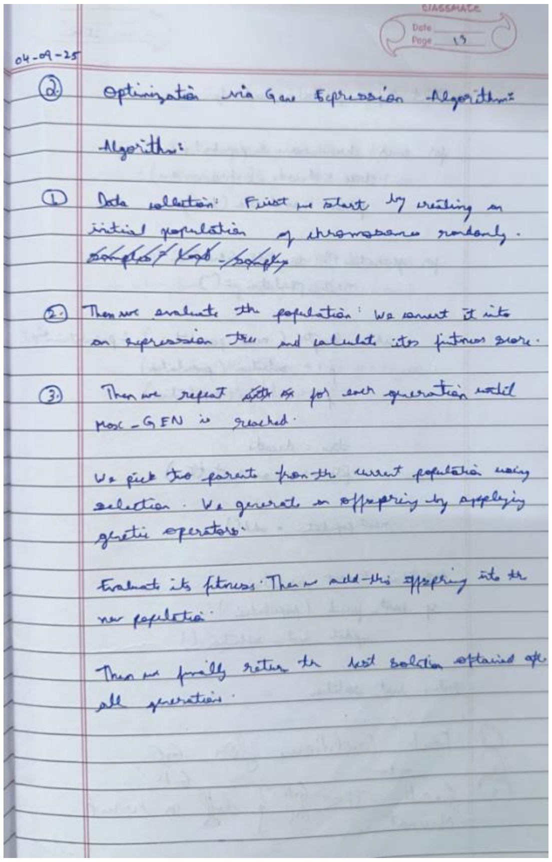
Output:

```
Gen 42 best avg delay: 50.00
Gen 43 best avg delay: 50.00
Gen 44 best avg delay: 50.00
Gen 45 best avg delay: 50.00
Gen 46 best avg delay: 50.00
Gen 47 best avg delay: 50.00
Gen 48 best avg delay: 50.00
Gen 49 best avg delay: 50.00
Gen 50 best avg delay: 50.00

optimal signal plan:
cycle length: 58.40
north south green: 33.19
east west green: 25.20
average delay: 50.00
```

**Program 2**

Optimization via Gene Expression Algorithm

Algorithm:



Page 13

04-09-25

(2) Optimization via Gene Expression Algorithm:

Algorithm:

(1) Data selection: First we start by creating an initial population of chromosome randomly.

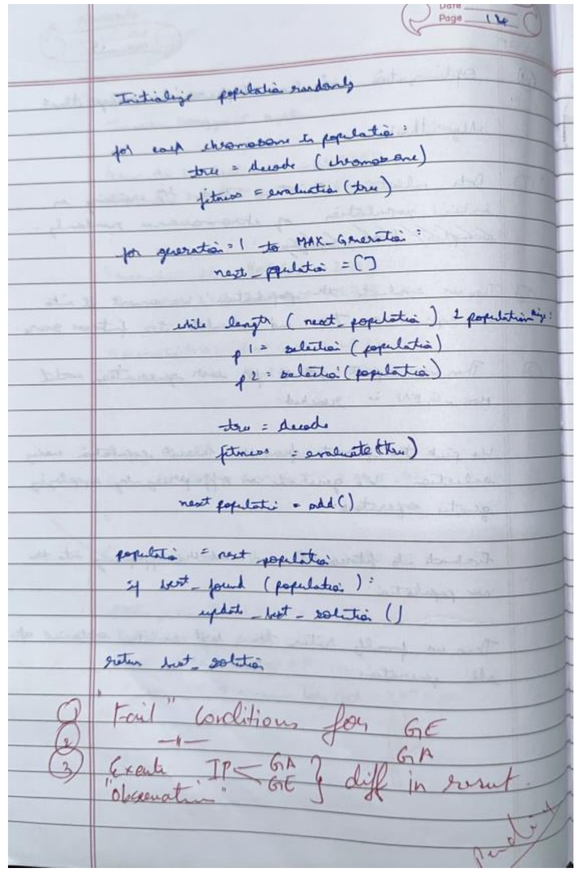
(2) Then we evaluate the population: We convert it into an expression tree and calculate its fitness score.

(3) Then we repeat for each generation until Max_GEN is reached.

We pick two parents from the current population using selection. We generate an offspring by applying genetic operators.

Evaluate its fitness. Then we add this offspring into the new population.

Then we finally return the best solution obtained after all generations.



Page 14

Initialize population randomly

```
for each chromosome in population:
        tree = decode (chromosome)
        fitness = evaluation (tree)

for generation = 1 to MAX_Generation:
        next_population = []

        while length ( next_population ) < population_size:
                p1 = selection (population)
                p2 = selection (population)

                tree = decode
                fitness = evaluate (tree)

        next population = add()

population = next population
 if best_found (population):
        update_best_solution ()

return best_solution
```

(1) "Fail" conditions for GE
(2) — " — GA
(3) Execute IP < GA, GE } diff in result
"Observation"

Code:

```
import random, math

POP=50
GEN=60
CHROM_LEN=20
FUNCTIONS=['+','-','*','/','sin','cos']
TERMINALS=['x']
ALL=FUNCTIONS+TERMINALS

def random_gene():
   return random.choice(ALL)

def valid_expression(expr,x):
   try:
      e=expr.replace('x',str(x))
      e=e.replace('/', '/(1e-6+')
      return eval(e)
```

```
    except:
        return 0

def fitness(expr,data):
    err=0
    for x,y in data:
        err+=abs(valid_expression(expr,x)-y)
    return -err

def mutate(expr):
    i=random.randrange(len(expr))
    return expr[:i]+random_gene()+expr[i+1:]

def crossover(a,b):
    i=random.randrange(1,len(a)-1)
    return a[:i]+b[i:], b[:i]+a[i:]

def random_expr():
    return ''.join(random_gene() for _ in range(CHROM_LEN))

def GEP(data):
    pop=[random_expr() for _ in range(POP)]
    for _ in range(GEN):
        fits=[fitness(e,data) for e in pop]
        new=[]
        elites=sorted(range(len(fits)), key=lambda i: fits[i], reverse=True)[:2]
        for i in elites:
            new.append(pop[i])
        while len(new)<POP:
            p1=random.choice(pop)
            p2=random.choice(pop)
            if random.random()<0.7:
                c1,c2=crossover(p1,p2)
            else:
                c1,c2=p1,p2
            if random.random()<0.3: c1=mutate(c1)
            if random.random()<0.3: c2=mutate(c2)
            new+= [c1,c2]
        pop=new[:POP]
    fits=[fitness(e,data) for e in pop]
    return pop[fits.index(max(fits))]

data=[(x,x*x+2*x+1) for x in range(-5,6)]
best=GEP(data)
print("Best:",best)
for x,y in data:
    print(x,y,valid_expression(best,x))
```

Output:

```
Generation 1: Best = 0.5120, Avg = 0.0670
Generation 2: Best = 0.5120, Avg = 0.1301
Generation 3: Best = 0.8134, Avg = 0.1283
Generation 4: Best = 0.8134, Avg = 0.1125
Generation 5: Best = 0.8134, Avg = 0.1040
Generation 6: Best = 0.8134, Avg = 0.1010
Generation 7: Best = 0.8134, Avg = 0.1171
Generation 8: Best = 0.8134, Avg = 0.1064
Generation 9: Best = 0.8134, Avg = 0.1085
Generation 10: Best = 0.8134, Avg = 0.1150

Best value found: 42.229387766752076
Fitness: 0.813413006900096
```

**Program 3**
Particle Swarm Optimization

Algorithm:

11-09-25

3 Particle Swarm optimization for function optimization:

Define the objective function general (minimize or maximize)

def objective_function (position)
    return f( position)

set PSO parameters (Initial with the parameters

num_particles = None
num_dimensions = None
max_iteration = None

C1 : (cognitive coeff)
C2 : (social coeff)

Initialize particles

particles = []
velocities = []
pbest = []
pbest_value = []

I.P

upper_bound_pos, lower_bound_pos

Page 19

upper bound - velo, lower bound_velo

for each particle : (Iterate)

randomly assign position within limits
randomly assign velocities
set personal best = Initial position
Then evaluate objective function at that position

Initialize global best (Best among all particles)

gbest = position of best particle so far
gbest_value = fitness value at gbest

for iteration in range (max_iteration)

for each particle:
If fitness > personal update pbest
If fitness > global update global

for each particle:
for each dimension d:
update velocity
v = w * v + c1 * r1 (pbest - pos)
+ c2 * r2 (gbest - curr_pos)

update position
x = curr_position + v
Ensure position and velocity stay within defined limits
Return gbest (best solution found)
gbest_value (its fitness)

Code:

```
import cv2
import numpy as np
import random
import math
from copy import deepcopy

IMAGE_PATH = "image.jpg"
NUM_PARTICLES = 30
ITERATIONS = 100
W = 0.7
C1 = 1.5
C2 = 1.5
```

```
LOW_MIN, LOW_MAX = 10.0, 150.0
HIGH_MIN, HIGH_MAX = 50.0, 300.0
DIL_MIN, DIL_MAX = 0.0, 6.0
MIN_CONTOUR_AREA = 100.0

random.seed(42)
np.random.seed(42)

def load_image_gray(path):
    img = cv2.imread(path)
    if img is None:
        raise FileNotFoundError(f"Cannot load image: {path}. Put image file named '{path}' in working
directory.")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img, gray

def eval_params_on_image(gray, params):
    low, high, dilf = params
    low = float(np.clip(low, LOW_MIN, LOW_MAX))
    high = float(np.clip(high, HIGH_MIN, HIGH_MAX))
    if high <= low + 1.0:
        high = min(HIGH_MAX, low + 1.0)
    dil_iter = int(round(np.clip(dilf, DIL_MIN, DIL_MAX)))
    blur = cv2.GaussianBlur(gray, (5, 5), 0)
    edges = cv2.Canny(blur, int(low), int(high))
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
    deep_edges = cv2.dilate(edges, kernel, iterations=dil_iter)
    cnts = cv2.findContours(deep_edges.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    contours = cnts[0] if len(cnts) == 2 else cnts[1]
    total_area = 0.0
    valid = []
    for c in contours:
        a = cv2.contourArea(c)
        if a >= MIN_CONTOUR_AREA:
            total_area += a
            valid.append(c)
    return total_area, edges, deep_edges, valid

def init_particles(n):
    particles = []
    velocities = []
    for _ in range(n):
        low = random.uniform(LOW_MIN, LOW_MAX)
        high = random.uniform(HIGH_MIN, HIGH_MAX)
        dil = random.uniform(DIL_MIN, DIL_MAX)
        particles.append(np.array([low, high, dil], dtype=float))
        velocities.append(np.array([random.uniform(-1,1), random.uniform(-1,1), random.uniform(-
0.5,0.5)], dtype=float))
```

```
    return particles, velocities

def clip_particle(p):
    p[0] = np.clip(p[0], LOW_MIN, LOW_MAX)
    p[1] = np.clip(p[1], HIGH_MIN, HIGH_MAX)
    p[2] = np.clip(p[2], DIL_MIN, DIL_MAX)
    if p[1] <= p[0] + 1.0:
        p[1] = min(HIGH_MAX, p[0] + 1.0)
    return p

def pso_optimize(gray, num_particles=NUM_PARTICLES, iterations=ITERATIONS):
    particles, velocities = init_particles(num_particles)
    pbest = [p.copy() for p in particles]
    pbest_score = [-math.inf] * num_particles
    gbest = None
    gbest_score = -math.inf

    for i in range(num_particles):
        score, _, _, _ = eval_params_on_image(gray, particles[i])
        pbest_score[i] = score
        if score > gbest_score:
            gbest_score = score
            gbest = particles[i].copy()

    for t in range(1, iterations+1):
        for i in range(num_particles):
            r1 = random.random()
            r2 = random.random()
            velocities[i] = (W * velocities[i] +
                             C1 * r1 * (pbest[i] - particles[i]) +
                             C2 * r2 * (gbest - particles[i]))
            particles[i] = particles[i] + velocities[i]
            particles[i] = clip_particle(particles[i])

            score, _, _, _ = eval_params_on_image(gray, particles[i])

            if score > pbest_score[i]:
                pbest_score[i] = score
                pbest[i] = particles[i].copy()
                if score > gbest_score:
                    gbest_score = score
                    gbest = particles[i].copy()

        print(f"Iteration {t} best fitness: {round(gbest_score,4)}")

    best_area, best_edges, best_deep, best_contours = eval_params_on_image(gray, gbest)
    return gbest, best_area, best_edges, best_deep, best_contours

if __name__ == "__main__":
```

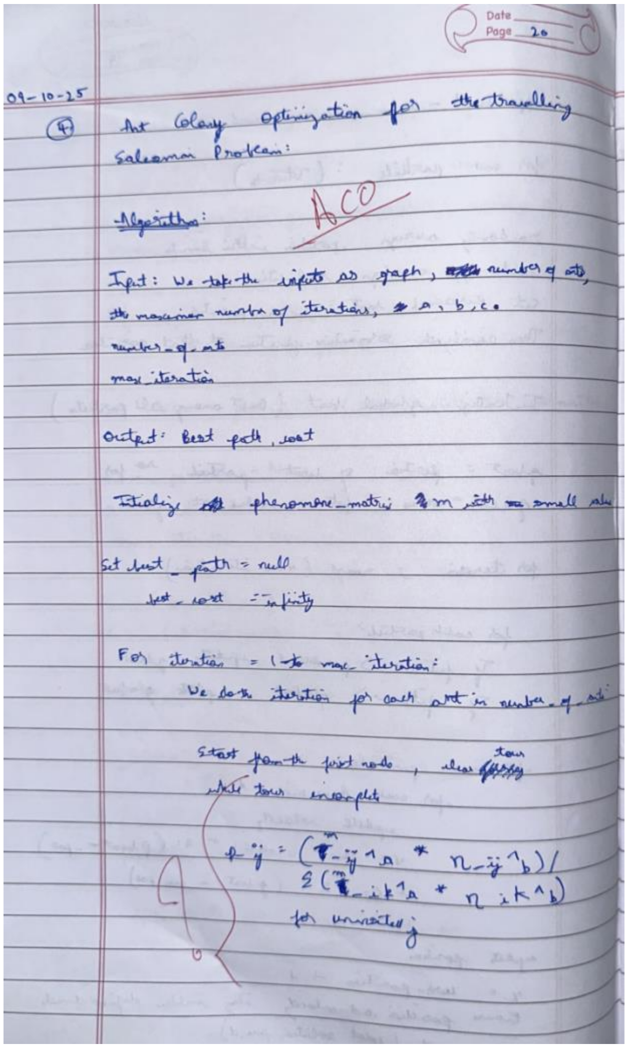
```
    img_color, img_gray = load_image_gray(IMAGE_PATH)
    best_params, best_score, best_edges, best_deep, best_contours = pso_optimize(img_gray,
NUM_PARTICLES, ITERATIONS)

    low, high, dil = best_params
    print("\nBest parameters found:")
    print(f"low threshold: {low:.2f}")
    print(f"high threshold: {high:.2f}")
    print(f"dilation iterations: {int(round(dil))}")
    print(f"best fitness (total contour area): {round(best_score,4)}")

    boxed = img_color.copy()
    for i, c in enumerate(best_contours):
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(boxed, (x,y), (x+w, y+h), (0,255,0), 2)
        cv2.putText(boxed, f"Obj {i+1}", (x, y-6), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,255,0), 1)

    deep_overlay = img_color.copy()
    mask = best_deep > 0
    deep_overlay[mask] = [255,255,255]

    cv2.imwrite("pso_best_edges.png", best_edges)
    cv2.imwrite("pso_best_deep.png", best_deep)
    cv2.imwrite("pso_identified_boxes.png", boxed)
    cv2.imwrite("pso_deep_overlay.png", deep_overlay)

    print("\nSaved: pso_best_edges.png, pso_best_deep.png, pso_identified_boxes.png,
pso_deep_overlay.png")
```

Output:

```
Iteration 86 best fitness: 50.51135
Iteration 87 best fitness: 50.51135
Iteration 88 best fitness: 50.51135
Iteration 89 best fitness: 50.51135
Iteration 90 best fitness: 50.51135
Iteration 91 best fitness: 50.51135
Iteration 92 best fitness: 50.51135
Iteration 93 best fitness: 50.51135
Iteration 94 best fitness: 50.51135
Iteration 95 best fitness: 50.51135
Iteration 96 best fitness: 50.51135
Iteration 97 best fitness: 50.51135
Iteration 98 best fitness: 50.51135
Iteration 99 best fitness: 50.51135
Iteration 100 best fitness: 50.51135
Best Fitnessfactor found:1.0
Best Fitness Found:50.51135
```

**Program 4**
Ant Colony Optimization

Algorithm:


Date
Page 20

09-10-25

(4) Ant Colony Optimization for the travelling Salesman Problem:

ACO

Algorithm:

Input: We take the inputs as graph, number of ants, the maximum number of iterations, a, b, c.
number_of_ants
max_iteration

Output: Best path, cost

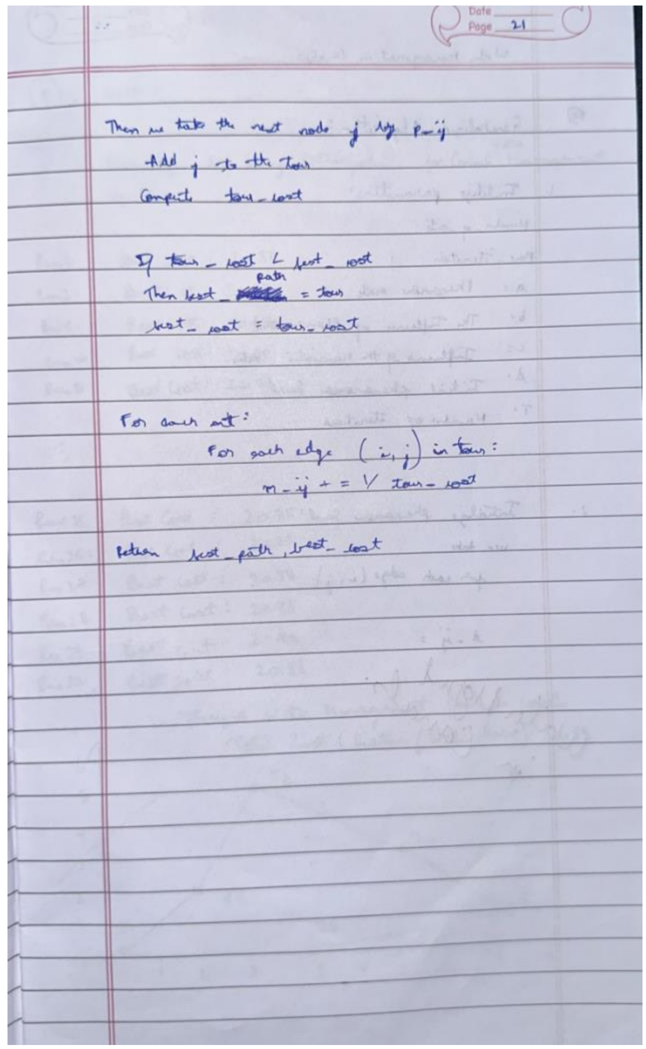
Initialize phenomene_matrix m with small value

Set best_path = null
best_cost = infinity

For iteration = 1 to max_iteration:
We do the iteration for each ant in number_of_ants

Start from the first node, clear tour
while tour incomplete

p_ij = (m_ij ^a * η_ij ^b) /
Σ (m_ik ^a * η ik ^b)
for unvisited j


Date
Page 21

Then we take the next node j by p_ij
Add j to the Tour
Compute tour_cost

If tour_cost < best_cost
Then best_path = tour
best_cost = tour_cost

For each ant:
For each edge (i, j) in tour:
m_ij += 1/ tour_cost

Return best_path, best_cost

Code:

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

coords = np.array([
    [0, 0],
    [3, 5],
    [6, 4],
    [8, 1],
    [4, 0],
    [2, 2]
])
```

```
def distance_matrix(coords):
    n = len(coords)
    dist = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            dist[i, j] = np.linalg.norm(coords[i] - coords[j])
    return dist

dist_matrix = distance_matrix(coords)

class ACO:
    def __init__(self, dist, ants=20, iterations=30, alpha=1, beta=5, evap=0.5, Q=100):
        self.dist = dist
        self.n = len(dist)
        self.pher = np.ones((self.n, self.n))
        self.ants = ants
        self.iter = iterations
        self.alpha = alpha
        self.beta = beta
        self.evap = evap
        self.Q = Q
        self.best_cost = float("inf")
        self.best_route = None

    def run(self):
        for it in range(self.iter):
            routes = []
            costs = []
            for _ in range(self.ants):
                r = self.build_route()
                c = self.cost(r)
                routes.append(r)
                costs.append(c)
                if c < self.best_cost:
                    self.best_cost = c
                    self.best_route = r

            self.update(routes, costs)
            print(f"Iteration {it + 1} | Best Cost: {self.best_cost:.2f}")

    def build_route(self):
        route = [random.randrange(self.n)]
        while len(route) < self.n:
            current = route[-1]
            probs = self.probabilities(current, route)
            route.append(np.random.choice(self.n, p=probs))
        return route

    def probabilities(self, current, visited):
```

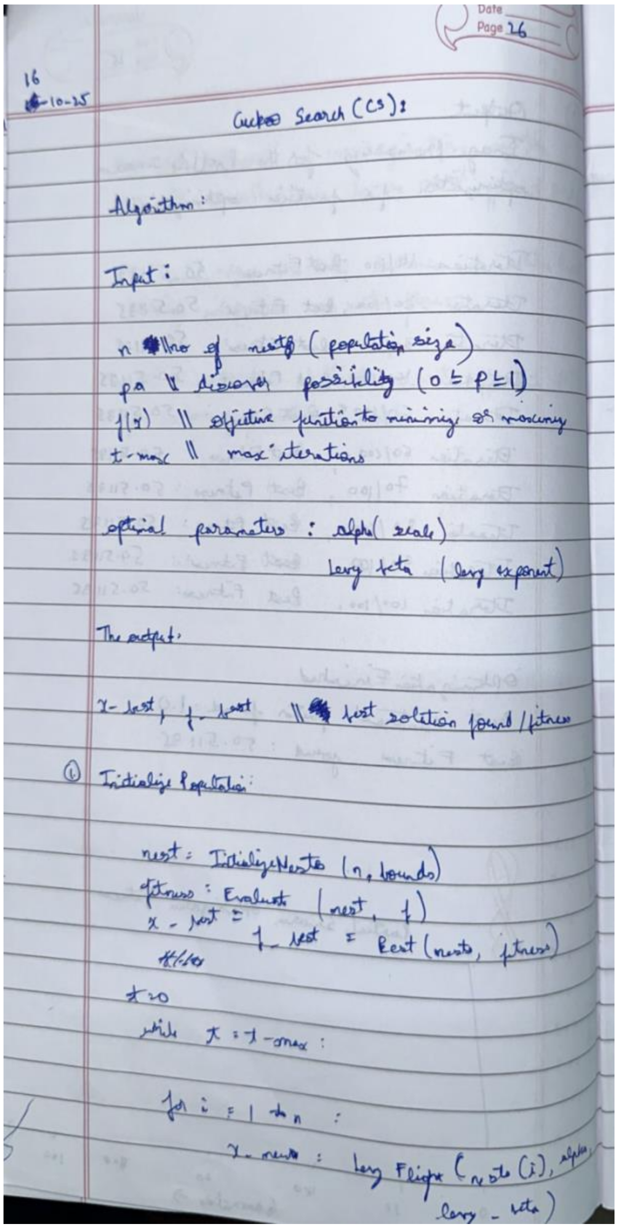
```
        p = np.zeros(self.n)
        for j in range(self.n):
            if j not in visited:
                p[j] = (self.pher[current, j]**self.alpha) * (1 / (self.dist[current, j] + 1e-10))**self.beta
        return p / p.sum()

    def cost(self, route):
        return sum(self.dist[route[i], route[(i+1) % self.n]] for i in range(self.n))

    def update(self, routes, costs):
        self.pher *= (1 - self.evap)
        for r, c in zip(routes, costs):
            for i in range(self.n):
                a, b = r[i], r[(i+1) % self.n]
                self.pher[a, b] += self.Q / c

aco = ACO(dist_matrix)
aco.run()

best_route = aco.best_route + [aco.best_route[0]]

plt.figure(figsize=(7, 5))
plt.scatter(coords[:, 0], coords[:, 1], s=80, c="blue")

for i in range(len(best_route) - 1):
    a, b = best_route[i], best_route[i+1]
    plt.plot([coords[a, 0], coords[b, 0]],
             [coords[a, 1], coords[b, 1]], "r-", linewidth=2)

for i, (x, y) in enumerate(coords):
    plt.text(x + 0.1, y + 0.1, f"J{i+1}")

plt.title(f"Optimized Route\nTotal Cost: {aco.best_cost:.2f}")
plt.grid(True)
plt.show()
```

Output:

```
Iteration 15 | Best Cost: 20.88
Iteration 16 | Best Cost: 20.88
Iteration 17 | Best Cost: 20.88
Iteration 18 | Best Cost: 20.88
Iteration 19 | Best Cost: 20.88
Iteration 20 | Best Cost: 20.88
Iteration 21 | Best Cost: 20.88
Iteration 22 | Best Cost: 20.88
Iteration 23 | Best Cost: 20.88
Iteration 24 | Best Cost: 20.88
Iteration 25 | Best Cost: 20.88
Iteration 26 | Best Cost: 20.88
Iteration 27 | Best Cost: 20.88
Iteration 28 | Best Cost: 20.88
Iteration 29 | Best Cost: 20.88
Iteration 30 | Best Cost: 20.88
```

**Program 5**
Cuckoo Search

Algorithm:



Page 26

16-10-25

Cuckoo Search (CS):

Algorithm:

Input:

n // no of nests (population size)
pa // discover possibility (0 ≤ p ≤ 1)
f(x) // objective function to minimize or maximize
t-max // max iterations

optimal parameters : alpha (scale)
Levy beta (levy exponent)

The output:

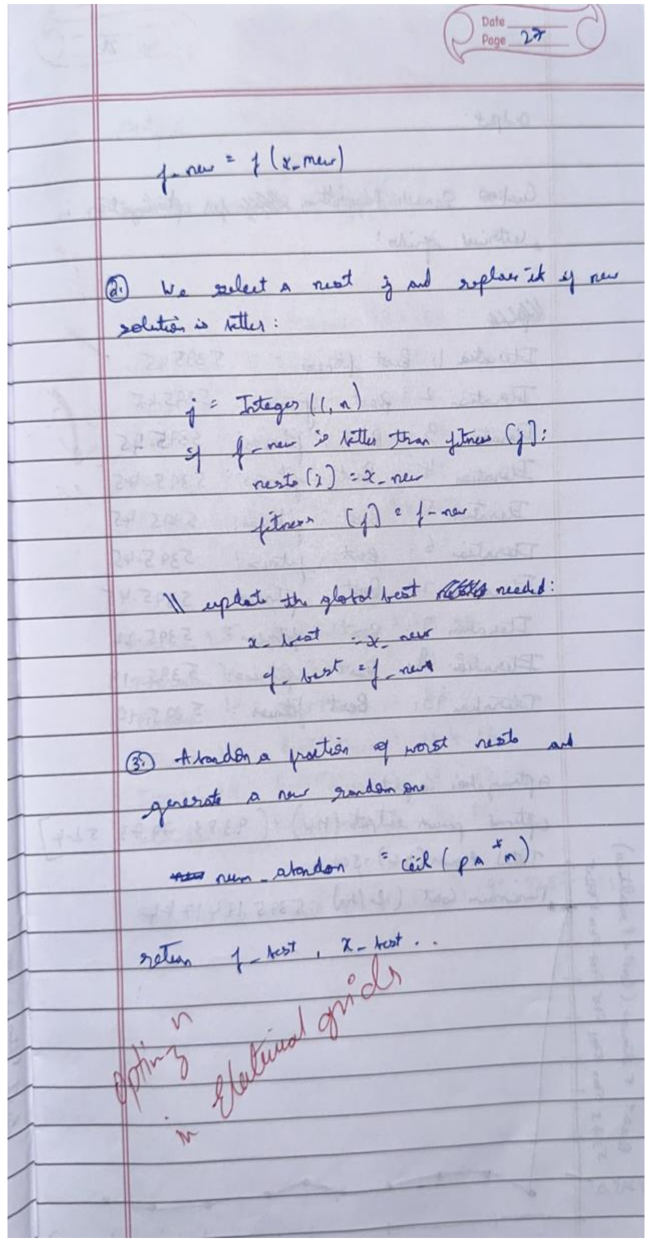
x-best, f-best // best solution found / fitness

1. Initialize Population:

nest = InitializeNests (n, bounds)
fitness = Evaluate (nest, f)
x-best = f-best = Best (nests, fitness)

t=0
while t = t-max :

for i = 1 to n :
x-new = levy Flight (nests (i), alpha, levy-beta)



Page 27

f-new = f (x-new)

2. We select a nest j and replace it if new solution is better:

j = Integer (1, n)
if f-new is better than fitness (j):
nests (j) = x-new
fitness (j) = f-new

// update the global best if needed:
x-best = x-new
f-best = f-new

3. Abandon a fraction of worst nests and generate a new random one

num-abandon = ceil (pa * n)

return f-best, x-best . .

Optimizing electrical grids

Code:

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def power_consumption(x):
    return np.sum(0.8 * x**2 + 2*np.sin(x) + 10)
```

```
n = 20
max_iter = 30
pa = 0.25
lb, ub = -10, 10

nests = np.random.uniform(lb, ub, (n, 5))
fitness = np.array([power_consumption(x) for x in nests])

def get_best(nests, fitness):
    idx = np.argmin(fitness)
    return nests[idx], fitness[idx]

best_nest, best_power = get_best(nests, fitness)
power_history = []

for t in range(max_iter):
    new_nests = np.copy(nests)
    for i in range(n):
        step = np.random.randn(*nests[i].shape)
        new_nests[i] = nests[i] + 0.01 * step * (nests[i] - best_nest)
        new_nests[i] = np.clip(new_nests[i], lb, ub)

    new_fitness = np.array([power_consumption(x) for x in new_nests])

    for i in range(n):
        if new_fitness[i] < fitness[i]:
            fitness[i] = new_fitness[i]
            nests[i] = new_nests[i]

    abandon_mask = np.random.rand(n, 5) < pa
    stepsize = np.random.rand() * (nests[np.random.permutation(n)] -
nests[np.random.permutation(n)])
    nests = nests + stepsize * abandon_mask
    nests = np.clip(nests, lb, ub)

    fitness = np.array([power_consumption(x) for x in nests])
    best_nest, best_power = get_best(nests, fitness)
    power_history.append(best_power)

    print(f"Iteration {t+1} : Power (MW) : {best_power:.4f}")

optimal_power = np.round(best_power, 4)
print("Optimization Complete")
print(f"Minimum Power used (Best fitness): {optimal_power} MW")

plt.plot(range(1, max_iter+1), power_history, marker='o', color='orange')
plt.title("Cuckoo Search Optimization - Minimum Power Usage (MW)")
plt.xlabel("Iteration")
```

```
plt.ylabel("Power (MW)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Output:

```
Iteration 14 : Power (MW) : 151.7949
Iteration 15 : Power (MW) : 151.7949
Iteration 16 : Power (MW) : 151.7949
Iteration 17 : Power (MW) : 134.6459
Iteration 18 : Power (MW) : 134.6459
Iteration 19 : Power (MW) : 80.2864
Iteration 20 : Power (MW) : 83.9449
Iteration 21 : Power (MW) : 84.2252
Iteration 22 : Power (MW) : 79.5543
Iteration 23 : Power (MW) : 79.1882
Iteration 24 : Power (MW) : 79.1882
Iteration 25 : Power (MW) : 73.7510
Iteration 26 : Power (MW) : 82.9971
Iteration 27 : Power (MW) : 83.1182
Iteration 28 : Power (MW) : 144.1699
Iteration 29 : Power (MW) : 132.1251
Iteration 30 : Power (MW) : 96.8857
```

**Program 6**
Grey Wolf Optimization

Algorithm:

Date
Page 31

23-10-25

Grey Wolf optimizer (GWO):

Algorithm:

1. Define the function f(x).
2. Initialize Parameters:
   N = No of wolves
   max_iter = maximum number of iterations
3. Initialize the position of N wolves randomly within search space bounds.
4. Evaluate the fitness of each wolf using f(x).
5. We select 3 best wolves:
   Alpha = Best wolf
   Beta = Second ~~wolf~~ Best wolf
   Delta = Third Best wolf
6. for t = 1 to max_iteration:

identification & deepning

IP image outline

9

   for each wolf i:

   Generate random numbers r1, r2 ∈ [0, 1]
   Compute:
   $A = 2 * A * r1 - A$
   $C = 2 * r2$

Code:

```
import cv2
import numpy as np
import random
from google.colab import files

print("Please upload your image (click the Choose Files button)...")
uploaded = files.upload()
```

```python
if len(uploaded) == 0:
    raise SystemExit("No file uploaded. Re-run the cell and upload an image file (jpg/png).")

image_path = list(uploaded.keys())[0]

POPULATION = 20
MAX_ITER = 30
SEED = 42
random.seed(SEED)
np.random.seed(SEED)

LOW_MIN, LOW_MAX = 10.0, 150.0
HIGH_MIN, HIGH_MAX = 50.0, 300.0
DIL_MIN, DIL_MAX = 0.0, 6.0
MIN_CONTOUR_AREA = 100.0

def load_image_gray(path):
    img = cv2.imread(path)
    if img is None:
        raise FileNotFoundError(f"Cannot load image: {path}")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img, gray

def eval_params_on_image(gray, params):
    low, high, dilf = params
    low = float(np.clip(low, LOW_MIN, LOW_MAX))
    high = float(np.clip(high, HIGH_MIN, HIGH_MAX))
    if high <= low + 1.0:
        high = min(HIGH_MAX, low + 1.0)
    dil_iter = int(round(np.clip(dilf, DIL_MIN, DIL_MAX)))
    blur = cv2.GaussianBlur(gray, (5,5), 0)
    edges = cv2.Canny(blur, low, high)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
    deep_edges = cv2.dilate(edges, kernel, iterations=dil_iter)
    # handle findContours return (OpenCV 3 vs 4)
    cnts = cv2.findContours(deep_edges.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    contours = cnts[0] if len(cnts) == 2 else cnts[1]
    total_area = 0.0
    for c in contours:
        a = cv2.contourArea(c)
        if a >= MIN_CONTOUR_AREA:
            total_area += a
    return total_area

def initialize_wolves(n):
    wolves = []
    for _ in range(n):
        low = random.uniform(LOW_MIN, LOW_MAX)
```

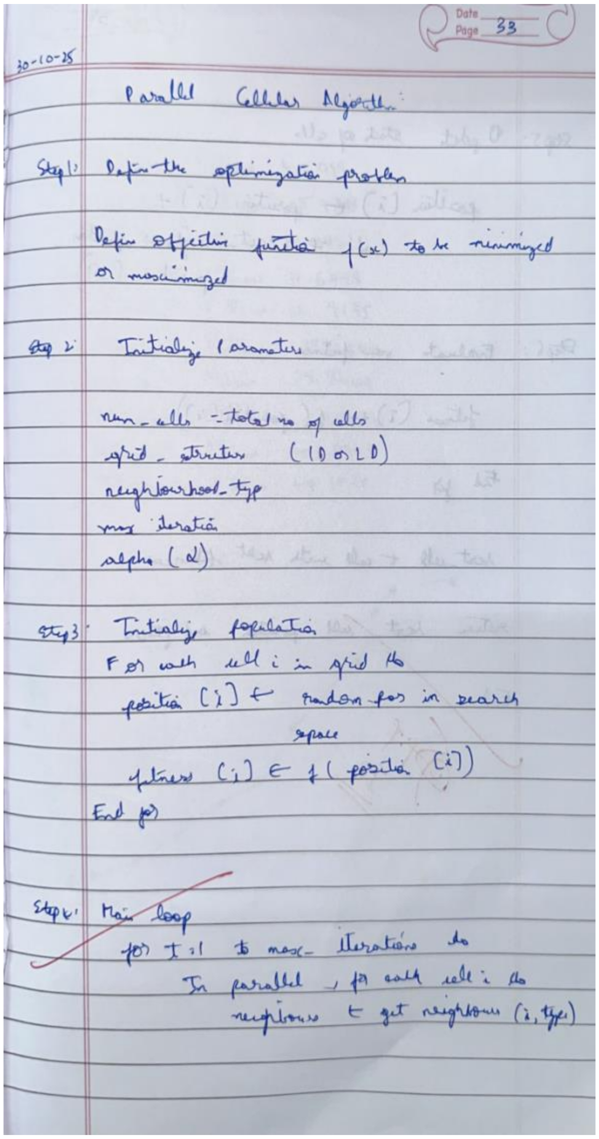
```
        high = random.uniform(HIGH_MIN, HIGH_MAX)
        dil = random.uniform(DIL_MIN, DIL_MAX)
        wolves.append(np.array([low, high, dil], dtype=float))
    return wolves

def gwo_optimize(gray, pop=POPULATION, iterations=MAX_ITER):
    wolves = initialize_wolves(pop)
    fitnesses = [eval_params_on_image(gray, w) for w in wolves]
    idx = sorted(range(len(wolves)), key=lambda i: fitnesses[i], reverse=True)
    alpha, beta, delta = wolves[idx[0]].copy(), wolves[idx[1]].copy(), wolves[idx[2]].copy()
    for t in range(iterations):
        a = 2.0 * (1.0 - (t / float(iterations)))
        for i in range(len(wolves)):
            X = wolves[i].copy()
            for leader in (alpha, beta, delta):
                r1, r2 = random.random(), random.random()
                A = 2.0 * a * r1 - a
                C = 2.0 * r2
                D = np.abs(C * leader - X)
                X = leader - A * D
            X[0] = np.clip(X[0], LOW_MIN, LOW_MAX)
            X[1] = np.clip(X[1], HIGH_MIN, HIGH_MAX)
            X[2] = np.clip(X[2], DIL_MIN, DIL_MAX)
            wolves[i] = X
        fitnesses = [eval_params_on_image(gray, w) for w in wolves]
        idx = sorted(range(len(wolves)), key=lambda i: fitnesses[i], reverse=True)
        alpha = wolves[idx[0]].copy()
        alpha_score = fitnesses[idx[0]]
        beta = wolves[idx[1]].copy()
        delta = wolves[idx[2]].copy()
        print(f"Iteration {t+1} best fitness: {alpha_score:.2f}")
    return alpha, alpha_score

img_color, img_gray = load_image_gray(image_path)
best_params, best_score = gwo_optimize(img_gray)
```

Output:

```
Iteration 22 best fitness: 51.9681
Iteration 23 best fitness: 51.9681
Iteration 24 best fitness: 51.9681
Iteration 25 best fitness: 51.9681
Iteration 26 best fitness: 51.9681
Iteration 27 best fitness: 51.9681
Iteration 28 best fitness: 51.9681
Iteration 29 best fitness: 51.9681
Iteration 30 best fitness: 51.9681
```

**Program 7**
Parallel Cellular Algorithm

Algorithm:



Date 30-10-25 Page 33

Parallel Cellular Algorithm:

Step 1: Define the optimization problem

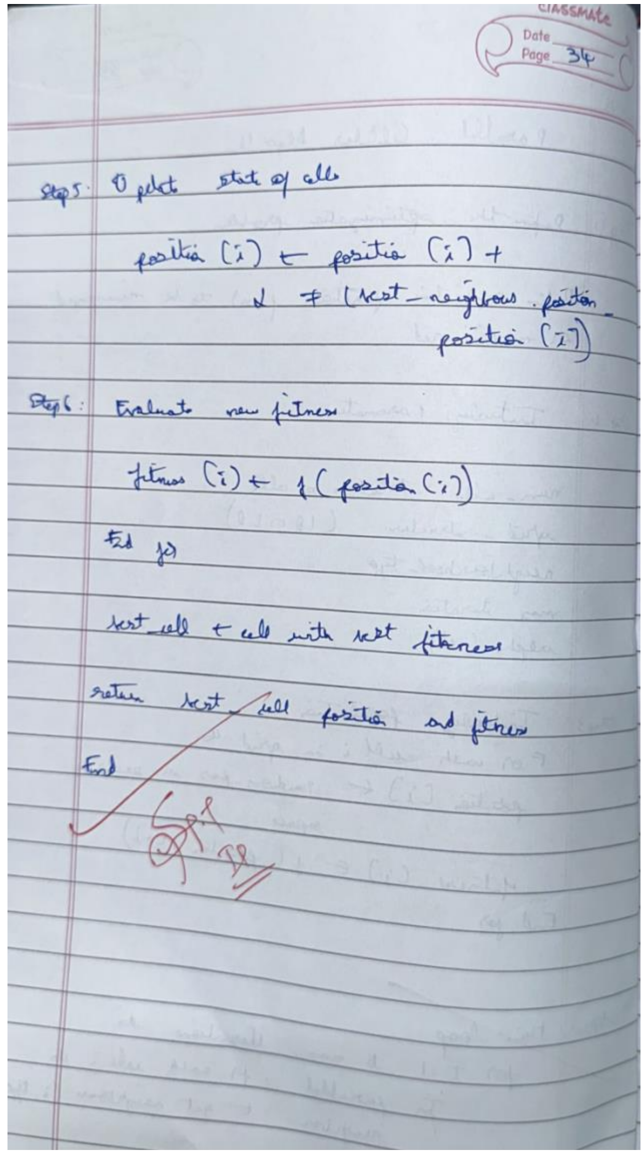
Define objective function f(x) to be minimized or maximized

Step 2: Initialize Parameters

num_cells - total no of cells
grid_structure (1D or 2D)
neighbourhood_type
max iteration
alpha (α)

Step 3: Initialize population
For each cell i in grid do
position [i] ← random pos in search space
fitness (i) ← f(position (i))
End for

Step 4: Main loop
for t = 1 to max_iterations do
In parallel, for each cell i do
neighbours ← get neighbours (i, type)



Page 34

Step 5: Update state of cells

position [i] ← position [i] + α * (best_neighbour_position - position [i])

Step 6: Evaluate new fitness

fitness (i) ← f(position (i))

End for

best_cell ← cell with best fitness

return best_cell position and fitness

End

Code:

```
import cv2
import numpy as np
import random
from copy import deepcopy

IMAGE_PATH = "image.jpg"
GRID_ROWS = 10
```

```
GRID_COLS = 10
N_ITER = 100
MUTATION_RATE = 0.3
SEED = 42

LOW_MIN, LOW_MAX = 10.0, 150.0
HIGH_MIN, HIGH_MAX = 50.0, 300.0
DIL_MIN, DIL_MAX = 0.0, 6.0
MIN_CONTOUR_AREA = 100.0

random.seed(SEED)
np.random.seed(SEED)

def load_image_gray(path):
    img = cv2.imread(path)
    if img is None:
        raise FileNotFoundError(f"Cannot load image: {path}")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img, gray

def eval_params_on_image(gray, params):
    low, high, dilf = params
    low = float(np.clip(low, LOW_MIN, LOW_MAX))
    high = float(np.clip(high, HIGH_MIN, HIGH_MAX))
    if high <= low + 1.0:
        high = min(HIGH_MAX, low + 1.0)
    dil_iter = int(round(np.clip(dilf, DIL_MIN, DIL_MAX)))
    blur = cv2.GaussianBlur(gray, (5,5), 0)
    edges = cv2.Canny(blur, int(low), int(high))
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
    deep_edges = cv2.dilate(edges, kernel, iterations=dil_iter)
    cnts = cv2.findContours(deep_edges.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    contours = cnts[0] if len(cnts) == 2 else cnts[1]
    total_area = 0.0
    valid = []
    for c in contours:
        a = cv2.contourArea(c)
        if a >= MIN_CONTOUR_AREA:
            total_area += a
            valid.append(c)
    return total_area, edges, deep_edges, valid

def random_params():
    low = random.uniform(LOW_MIN, LOW_MAX)
    high = random.uniform(HIGH_MIN, HIGH_MAX)
    dil = random.uniform(DIL_MIN, DIL_MAX)
    return [low, high, dil]
```

```
def neighbors_indices(r, c, rows, cols):
    neigh = []
    for dr in (-1, 0, 1):
        for dc in (-1, 0, 1):
            if dr == 0 and dc == 0:
                continue
            nr = (r + dr) % rows
            nc = (c + dc) % cols
            neigh.append((nr, nc))
    return neigh

def local_mutation(params):
    p = params[:]
    if random.random() < MUTATION_RATE:
        p[0] += random.gauss(0, 8.0)
    if random.random() < MUTATION_RATE:
        p[1] += random.gauss(0, 12.0)
    if random.random() < MUTATION_RATE:
        p[2] += random.gauss(0, 1.2)
    p[0] = float(np.clip(p[0], LOW_MIN, LOW_MAX))
    p[1] = float(np.clip(p[1], HIGH_MIN, HIGH_MAX))
    p[2] = float(np.clip(p[2], DIL_MIN, DIL_MAX))
    if p[1] <= p[0] + 1.0:
        p[1] = min(HIGH_MAX, p[0] + 1.0)
    return p

def parallel_cellular_image_processing(image_path):
    img_color, img_gray = load_image_gray(image_path)

    grid = [[random_params() for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
    grid_scores = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
    grid_edges = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
    grid_deep = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
    grid_contours = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]

    global_best_params = None
    global_best_score = -float("inf")

    for i in range(GRID_ROWS):
        for j in range(GRID_COLS):
            score, edges, deep_edges, conts = eval_params_on_image(img_gray, grid[i][j])
            grid_scores[i][j] = score
            grid_edges[i][j] = edges
            grid_deep[i][j] = deep_edges
            grid_contours[i][j] = conts
            if score > global_best_score:
                global_best_score = score
                global_best_params = deepcopy(grid[i][j])
```

```
    for it in range(1, N_ITER + 1):
        new_grid = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
        new_scores = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
        new_edges = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
        new_deep = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
        new_contours = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]

        for r in range(GRID_ROWS):
            for c in range(GRID_COLS):
                best_local_params = grid[r][c]
                best_local_score = grid_scores[r][c]

                for nr, nc in neighbors_indices(r, c, GRID_ROWS, GRID_COLS):
                    if grid_scores[nr][nc] > best_local_score:
                        best_local_score = grid_scores[nr][nc]
                        best_local_params = grid[nr][nc]

                child_params = local_mutation(best_local_params)
                child_score, child_edges, child_deep, child_conts = eval_params_on_image(img_gray,
child_params)

                if child_score > best_local_score:
                    new_grid[r][c] = child_params
                    new_scores[r][c] = child_score
                    new_edges[r][c] = child_edges
                    new_deep[r][c] = child_deep
                    new_contours[r][c] = child_conts
                else:
                    new_grid[r][c] = best_local_params[:]
                    new_scores[r][c] = best_local_score
                    new_edges[r][c] = grid_edges[r][c]
                    new_deep[r][c] = grid_deep[r][c]
                    new_contours[r][c] = grid_contours[r][c]

                if new_scores[r][c] > global_best_score:
                    global_best_score = new_scores[r][c]
                    global_best_params = deepcopy(new_grid[r][c])

        grid = new_grid
        grid_scores = new_scores
        grid_edges = new_edges
        grid_deep = new_deep
        grid_contours = new_contours

        print(f"Iteration {it} best fitness: {round(global_best_score, 4)}")

    best_score, best_edges, best_deep, best_contours = eval_params_on_image(img_gray,
global_best_params)
    return {
```

```
        "best_params": global_best_params,
        "best_score": best_score,
        "best_edges": best_edges,
        "best_deep": best_deep,
        "best_contours": best_contours,
        "img_color": img_color
    }

if __name__ == "__main__":
    result = parallel_cellular_image_processing(IMAGE_PATH)
    bp = result["best_params"]
    bs = result["best_score"]
    be = result["best_edges"]
    bd = result["best_deep"]
    bcont = result["best_contours"]
    img_color = result["img_color"]

    print("\nFinal best parameters:")
    print(f"low_thresh: {bp[0]:.2f}, high_thresh: {bp[1]:.2f}, dilation_iters: {int(round(bp[2]))}")
    print(f"Final best fitness: {bs:.4f}")

    boxed = img_color.copy()
    for i, c in enumerate(bcont):
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(boxed, (x, y), (x + w, y + h), (0, 255, 0), 2)
        cv2.putText(boxed, f"Obj {i+1}", (x, y - 6), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
1)

    deep_overlay = img_color.copy()
    mask = bd > 0
    deep_overlay[mask] = [255, 255, 255]

    cv2.imwrite("pc_best_edges.png", be)
    cv2.imwrite("pc_best_deep.png", bd)
    cv2.imwrite("pc_identified_boxes.png", boxed)
    cv2.imwrite("pc_deep_overlay.png", deep_overlay)

    print("\nSaved: pc_best_edges.png, pc_best_deep.png, pc_identified_boxes.png,
pc_deep_overlay.png")
```

Output:

```
Iteration 1: fitness: 4.5089
Iteration 2: fitness: 5.8333
Iteration 3: fitness: 7.9842
Iteration 4: fitness: 11.6798
Iteration 5: fitness: 17.9135
Iteration 6: fitness: 27.4544
Iteration 7: fitness: 38.8449
Iteration 8: fitness: 45.0404
Iteration 9: fitness: 47.9802
Iteration 10: fitness: 48.1760
```