VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

POTNURU ASHRITA (1BM23CS235)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug-2024 to Jan-2026

B.M.S. College of Engineering,
BullTempleRoad,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 Potnuru Ashrita ***(1BM23CS235),*** 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.

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

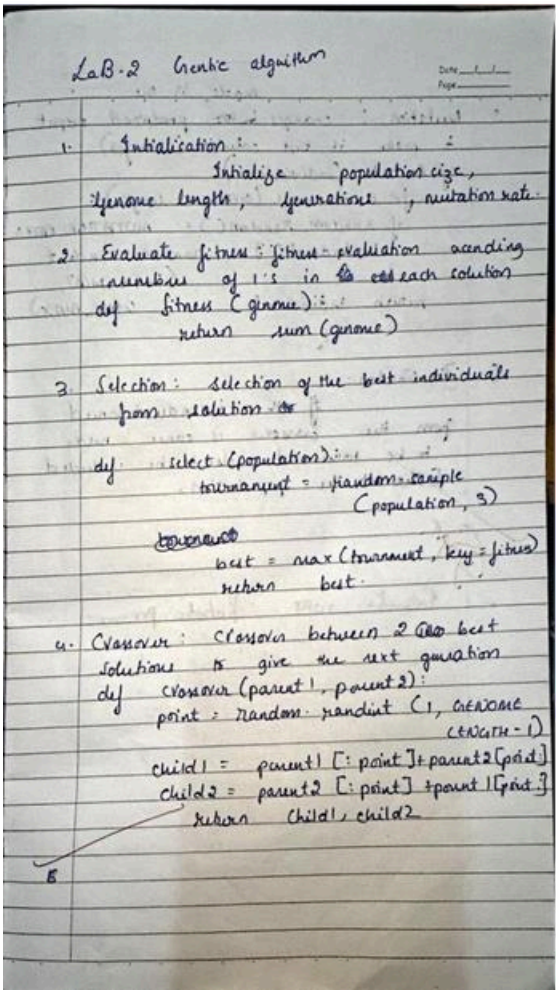
Index

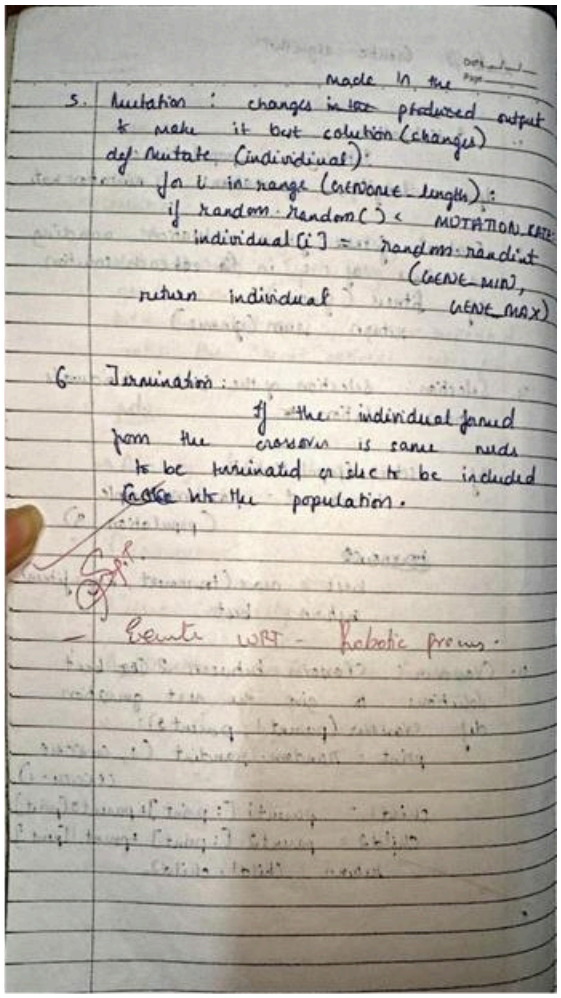
Sl. No.	*Date*	*ExperimentTitle*	*Page No.*
1	28-08-2025	Genetic Algorithm	4
2	04-09-2025	Gene Expression Algorithm	8
3	11-09-2025	Particle Swarm Optimisation	12
4	09-10-2025	Ant Colony Optimisation	16
5	16-10-2025	Cuckoo Search Optimisation	21
6	23-10-2025	Grey Wolf Optimisation	25
7	30-10-2025	Parallel Cellular Algorithm	34

GITHUB LINK : - https://github.com/potnuruashrita/1bm23cs235-BIS.git

Program 1: Genetic algorithm using Engineering design

Algorithm:





Code:

```
import random
from typing import List, Dict, Tuple


TASK_DURATIONS: Dict[str, float] = {
    "Pick": 3.0,
    "Place": 2.5,
    "Screw": 4.0,
    "Inspect": 2.0,
    "Label": 1.5,
    "Pack": 3.5,
}
SETUP_PENALTY: Dict[Tuple[str, str], float] = {
    ("Pick", "Place"): 0.4, ("Place", "Screw"): 0.8, ("Screw", "Inspect"): 0.6,
    ("Inspect", "Label"): 0.3, ("Label", "Pack"): 0.7,

    ("Pick", "Screw"): 1.2, ("Pick", "Inspect"): 0.9, ("Pick", "Label"): 0.9, ("Pick", "Pack"): 1.1,

    ("Place", "Inspect"): 0.5, ("Place", "Label"): 0.5, ("Place", "Pack"): 0.9,
    ("Screw", "Place"): 1.0, ("Screw", "Label"): 0.7, ("Screw", "Pack"): 1.0,
```

```
    ("Inspect","Place"):0.7,("Inspect","Pack"):0.8, ("Inspect", "Screw"): 0.9,
    ("Label","Inspect"):0.6,("Label","Screw"):0.9, ("Label", "Place"): 0.8, ("Label", "Pick"): 1.0,
    ("Pack","Label"):0.6,("Pack","Inspect"):0.9,("Pack", "Screw"): 1.2, ("Pack", "Place"): 1.0,
}
defschedule_cost(order:List[str]) -> float:
    """Totaltime=sumofdurations+sequence-dependent penalties."""
    total= 0.0
    for i, task in enumerate(order):
        total+= TASK_DURATIONS[task]
        ifi>0:
            prev=order[i- 1]
            total+=SETUP_PENALTY.get((prev,task), 0.0)
    returntotal


definit_population(tasks:List[str],size:int)->List[List[str]]:
    pop = []
    for _ in range(size):
        chrom= tasks[:]
        random.shuffle(chrom)
        pop.append(chrom)
    returnpop

deftournament_select(pop:List[List[str]],k:int)-> List[str]:
    """k-waytournamentselection(minimization)."""
    cand = random.sample(pop, k)
    cand.sort(key=schedule_cost)
    return cand[0][:]

deforder_crossover(p1:List[str],p2:List[str])->Tuple[List[str], List[str]]:
    """Order crossover (OX) for permutations."""
    n = len(p1)
    a, b = sorted(random.sample(range(n), 2))
    def ox(parent_a, parent_b):
        child = [None]*n
        child[a:b+1] = parent_a[a:b+1]

        fill = [g for g in parent_b if g not in child]
        j = 0
        for i in range(n):
            if child[i] is None:
                child[i] = fill[j]
                j+=1
        return child
    return ox(p1, p2), ox(p2, p1)

defswap_mutation(ch:List[str],rate:float)->None:
    if random.random() < rate:
        i, j = random.sample(range(len(ch)), 2)
```

```
        ch[i], ch[j] = ch[j], ch[i]

    defevolve(
        tasks: List[str],
        pop_size: int = 80,
        generations: int = 300,
        tournament_k: int = 4,
        crossover_rate: float = 0.9,
        mutation_rate: float = 0.2,
        elitism: int = 2,
        verbose:                bool=                True
    ):
        pop=init_population(tasks,pop_size)
        best = min(pop, key=schedule_cost)
        best_score = schedule_cost(best)
        if verbose:
            print(f"Gen0|best={best_score:.3f} s | {best}")

        for g in range(1, generations + 1):
            new_pop = []
            pop_sorted=sorted(pop,key=schedule_cost)
            new_pop.extend([ch[:]forchinpop_sorted[:elitism]])
            #Createoffspring
            while len(new_pop) < pop_size:

                p1=tournament_select(pop,tournament_k)
                p2=tournament_select(pop,tournament_k)
                ifrandom.random()<crossover_rate:
                c1,c2=order_crossover(p1,p2)
                else:
                    c1, c2 = p1[:], p2[:]
                swap_mutation(c1,mutation_rate)
                swap_mutation(c2,mutation_rate)
                new_pop.append(c1)
                if len(new_pop) < pop_size:
                    new_pop.append(c2)

            pop = new_pop
            cur_best=min(pop,key=schedule_cost)
            cur_score=schedule_cost(cur_best)
            if cur_score < best_score:
                best,best_score=cur_best,cur_score

            ifverboseand(g%25==0org== generations):
                print(f"Gen{g}|best={best_score:.3f} s | {best}")

        return best, best_score

    if __name__ == "__main__":
```

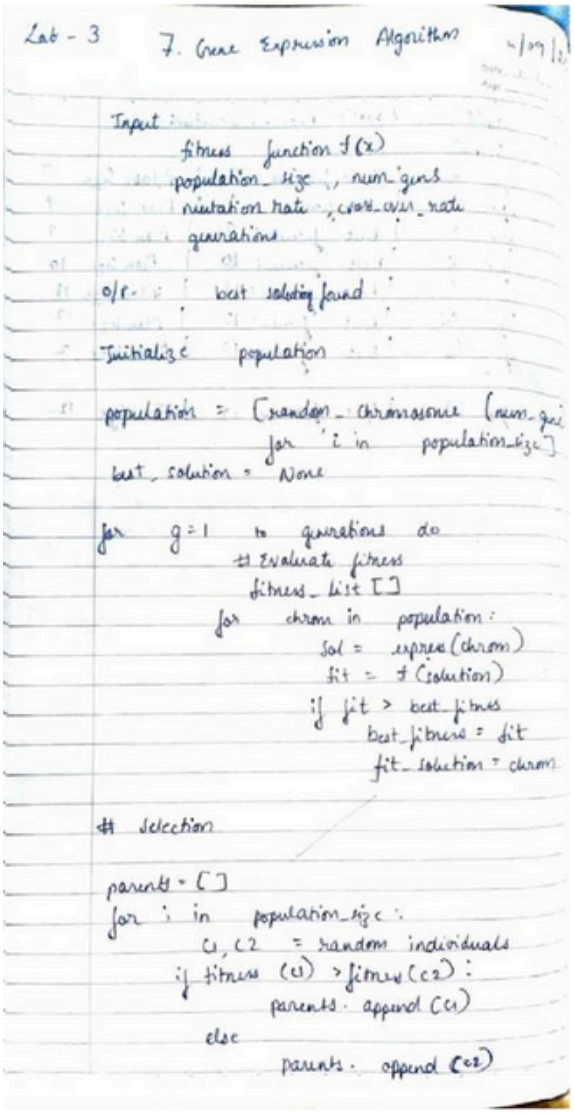
```
    random.seed(42)
    tasks = list(TASK_DURATIONS.keys())
    best_order, best_time = evolve(tasks, pop_size=120, generations=250, verbose=True)
    print("\nBest found sequence:")
    print(best_order)
print(f"Estimated total time: {best_time:.3f} s")
```

OUTPUT:

```
1BM23CS235-POTNURU ASHRITA
Gen 0 | best = 18.600 s | ['Pack', 'Pick', 'Place', 'Inspect', 'Label', 'Screw']
Gen 25 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 50 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 75 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 100 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 125 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 150 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 175 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 200 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 225 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 250 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']

Best found sequence:
['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Estimated total time: 18.400 s
```

Program 2: Genetic expression algorithm using Financial Forecasting

Algorithm:



Lab - 3 7. Gene Expression Algorithm

Input :
fitness function f(x)
population_size , num_gens
mutation_rate , cross_over_rate
generations

o/p : best solution found

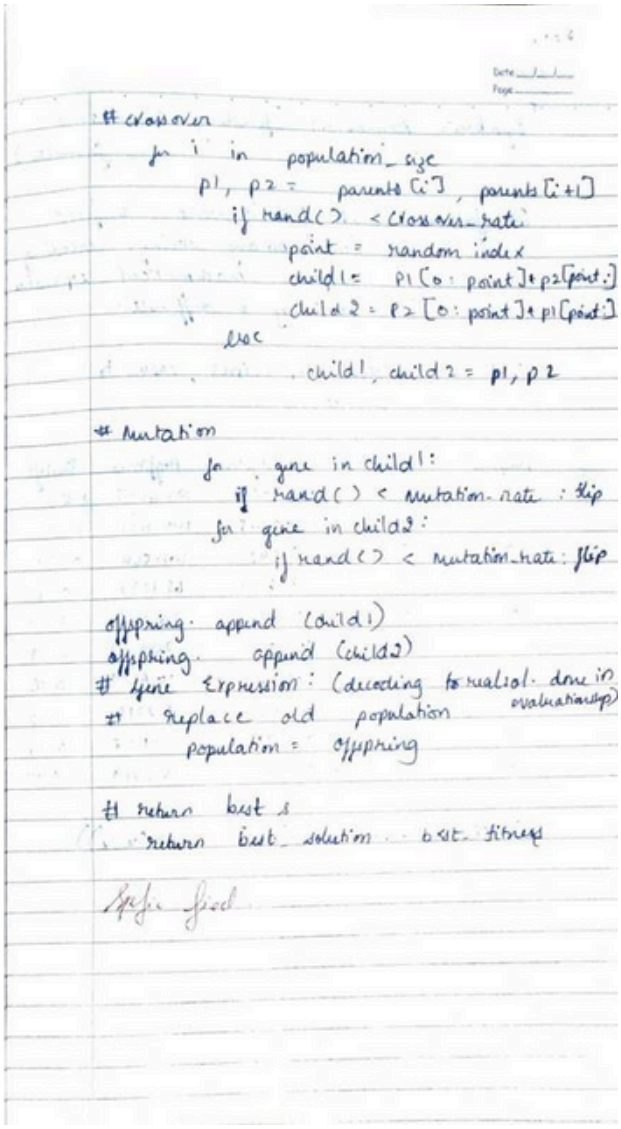
Initialize population

population = [random_chromosome (num_ge
for i in population_size]
best_solution = None

for g=1 to generations do
# Evaluate fitness
fitness_list []
for chrom in population:
sol = express(chrom)
fit = f(solution)
if fit > best_fitness
best_fitness = fit
fit_solution = chrom

Selection

parents = []
for i in population_size :
c1, c2 = random individuals
if fitness (c1) > fitness(c2) :
parents. append (c1)
else
parents. append (c2)



crossover
for i in population_size
p1, p2 = parents [i] , parents [i+1]
if rand() < cross_over_rate
point = random index
child1 = p1[0 : point] + p2[point:]
child 2 = p2 [0 : point] + p1[point:]
else
child1, child 2 = p1, p2

Mutation
for gene in child1 :
if rand() < mutation_rate : flip
for gene in child2 :
if rand() < mutation_rate : flip

offspring. append (child1)
offspring. append (child2)
# gene Expression : (decoding to realiol done in evaluation fn)
# replace old population
population = offspring

return best s
return best_solution, best_fitness

Specific find

Date __/__/____
Page _____

Symbolic Regression (finding mathematical formulas)

GA failure: GA fails it: works on fixed-length binary strings, encoding complex mathematical expressions directly is difficult.

GEP: evolves expressions, easier to encode.

Progress	Generation	Best fitness	Avg fitness	Time
1	0	168.9403	192.6539	0.36
2	1	121.5268	144.2123	0.33
3	2	79.2453	101.564	0.29
4	3	39.8321	65.1233	0.3
5	4	15.6234	36.8452	0.2
6	5	4.4932	20.2341	0.1
7	6	1.0461	10.6239	0.1
8	7	0.2034	5.2318	0.1
9	8	0.0100	2.1348	0.0
10	9	0.000	0.5219	0.0

Evation: add(add(x,1), mul(x,x))

Code:

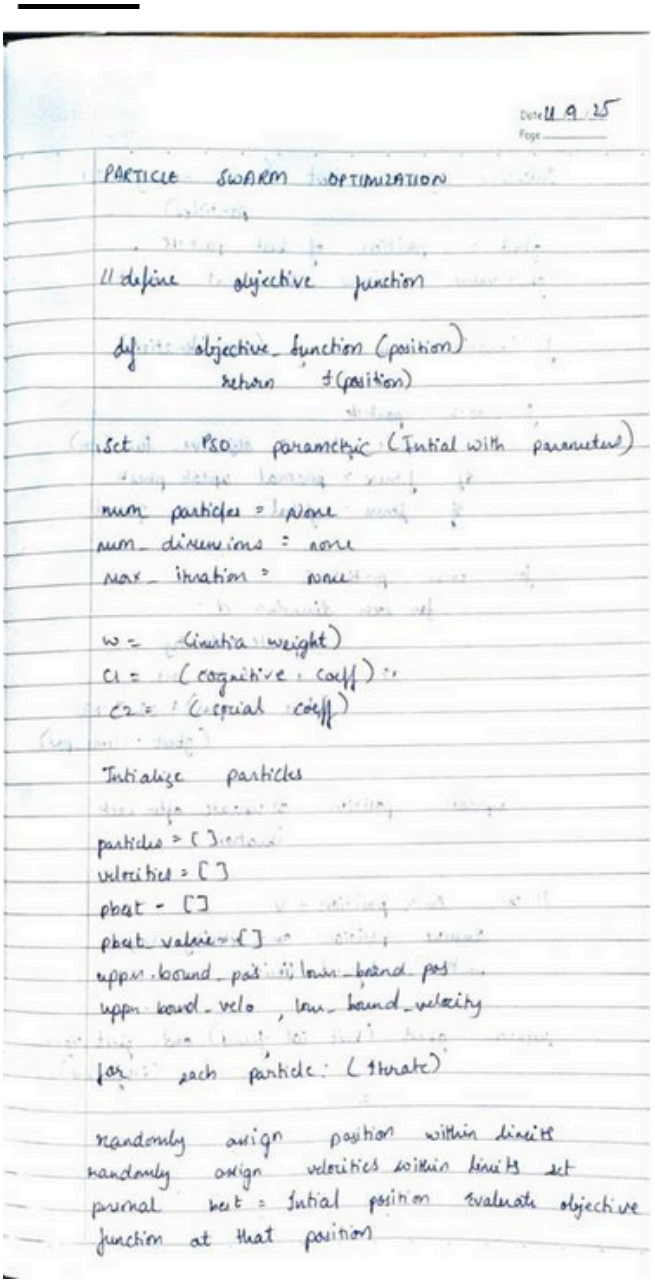
```
importnumpy as np
# Objective function (example: minimize sphere function)
def fitness(x):


    return np.sum(x**2)

# GA parameters
pop_size = 20
chrom_length = 5
generations = 50
mutation_rate = 0.1
# Initialization: random population in [-10, 10]
population = np.random.uniform(-10, 10, (pop_size, chrom_length))
for gen in range(generations):


    fitness_values = np.array([fitness(ind) for ind in population])
    # Selection (roulette wheel)
    fitness_inv = 1 / (fitness_values + 1e-6)
    probs = fitness_inv / np.sum(fitness_inv)
    parents_idx = np.random.choice(pop_size, size=pop_size, p=probs)
    parents = population[parents_idx]
    # Crossover (single point)
    offspring = []
    for i in range(0, pop_size, 2):

        p1, p2 = parents[i], parents[(i+1) % pop_size]
        point = np.random.randint(1, chrom_length)
        child1 = np.concatenate((p1[:point], p2[point:]))
        child2 = np.concatenate((p2[:point], p1[point:]))
        offspring.extend([child1, child2])
    offspring = np.array(offspring[:pop_size])
    # Mutation
    mutations = np.random.rand(pop_size, chrom_length) < mutation_rate
    mutation_values = np.random.uniform(-1, 1, (pop_size, chrom_length))
    offspring = offspring + mutations * mutation_values
    population = offspring



# Best solution
best_idx = np.argmin([fitness(ind) for ind in population])
print("Best solution:", population[best_idx])
print("Best fitness:", fitness(population[best_idx]))
```

OUTPUT: -

```
1BM23CS235-POTNURU ASHRITA
Best solution: [ 0.0301218   0.09859474 -0.13087039  0.01176386 -0.08049134]
Best fitness: 0.0343725495821183
```

Program 3: Particle Swarm Optimization for function optimization

Algorithm:



PARTICLE SWARM OPTIMIZATION

// define objective function

def objective_function (position)
return f(position)

Set PSO parameters (Initial with parameters)
num_particles = none
num_dimensions = none
max_iteration = none

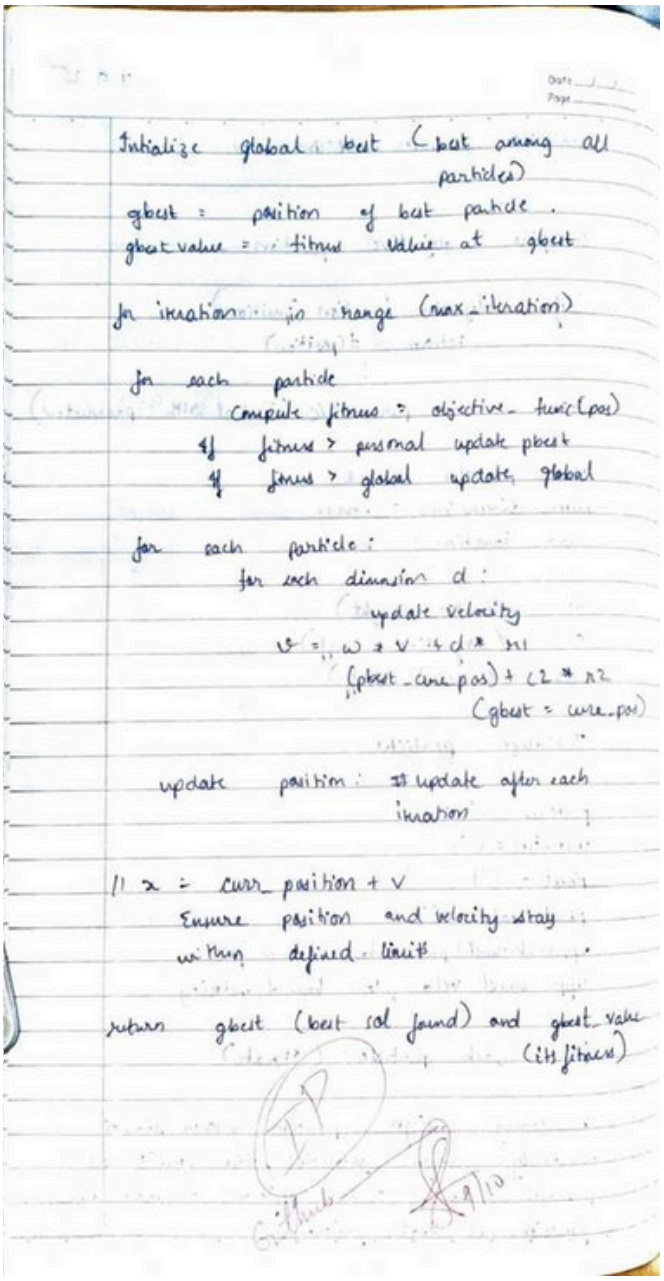
w = (inertia weight)
c1 = (cognitive coeff)
c2 = (social coeff)

Initialize particles

particles = [ ]
velocities = [ ]
pbest = [ ]
pbest_value = [ ]
upper_bound_pos , lower_bound_pos
upper_bound_velo , lower_bound_velocity

for each particle: (iterate)

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



Initialize global best (best among all particles)

gbest = position of best particle.
gbest_value = fitness value at gbest

for iteration in range (max_iteration)

for each particle
Compute fitness = objective_func(pos)
if fitness > personal update pbest
if fitness > global update global

for each particle:
for each dimension d:
update velocity
v = w * v + d * r1
(pbest_cur_pos) + c2 * r2
(gbest - cur_pos)

update position: # update after each iteration

// x = curr_position + v
Ensure position and velocity stay within defined limits

return gbest (best sol found) and gbest_value (its fitness)

Code: -

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

# Loadimageasgrayscale(replace'your_image.jpg' with your image file)
img=Image.open('your_image.jpg').convert('L')
img_np = np.array(img)

#Fitnessfunction:Otsu'sbetween-classvariance for threshold t
def otsu_fitness(t, image):


    t = int(t[0])
    if t <= 0 or t >= 255:
        return 0
    hist,_=np.histogram(image,bins=256,range=(0,256))
    total_pixels = image.size

    weight_bg=np.sum(hist[:t])/total_pixels
    weight_fg=np.sum(hist[t:])/total_pixels
    if weight_bg == 0 or weight_fg == 0:

        return 0

    mean_bg=np.sum(np.arange(t)*hist[:t]) / np.sum(hist[:t])
    mean_fg=np.sum(np.arange(t,256)*hist[t:]) / np.sum(hist[t:])
    between_class_variance=weight_bg*weight_fg * (mean_bg - mean_fg) ** 2
    return between_class_variance

# PSO parameters
num_particles = 30
max_iter = 50
w = 0.7     #inertiaweight
c1 = 1.5    #cognitivecoefficient
c2 = 1.5    #socialcoefficient

# Initializeparticlesrandomly(thresholdvalues between 0 and 255)
particles=np.random.uniform(0,255,(num_particles, 1))
velocities = np.zeros_like(particles)

# Initializepersonalbests
pbest_pos = particles.copy()
pbest_scores=np.array([otsu_fitness(p,img_np) for p in particles])
# Initializeglobalbest
gbest_idx = np.argmax(pbest_scores)
gbest_pos = pbest_pos[gbest_idx].copy()
gbest_score = pbest_scores[gbest_idx]
for iteration in range(max_iter):
for i in range(num_particles):

        r1 = np.random.rand()
        r2 = np.random.rand()
        velocities[i] = (w * velocities[i] +
                         c1*r1*(pbest_pos[i]-particles[i]) +
                         c2*r2*(gbest_pos-particles[i]))
```

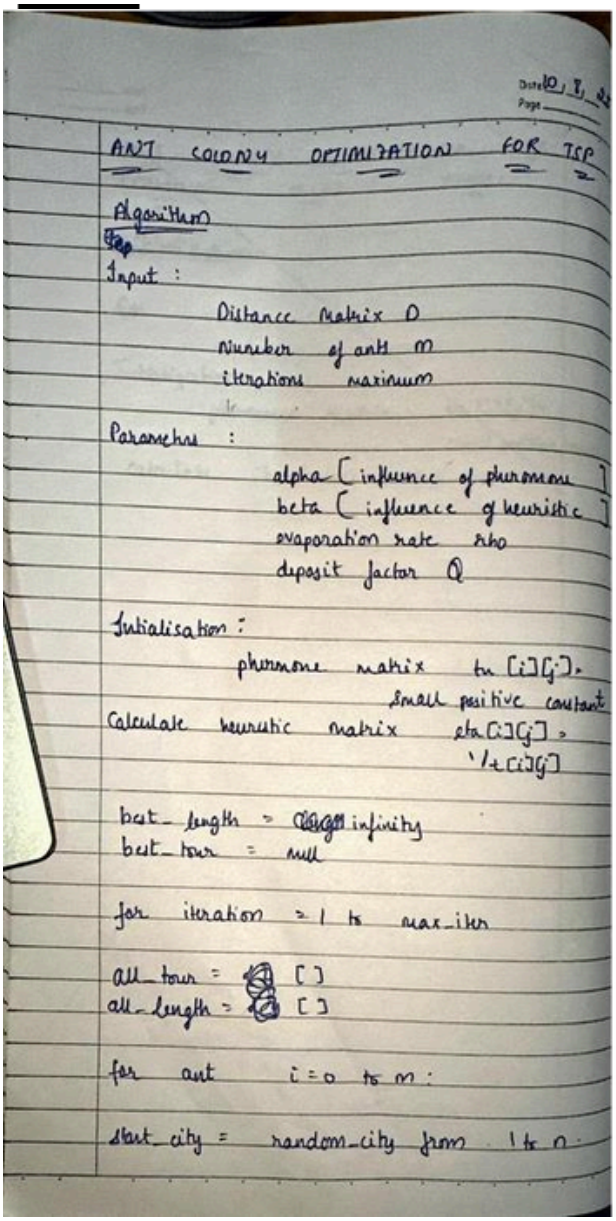
```
            particles[i] = particles[i] + velocities[i]
            particles[i] = np.clip(particles[i], 0, 255)

            fitness_val = otsu_fitness(particles[i], img_np)

            if fitness_val > pbest_scores[i]:

                pbest_scores[i] = fitness_val
                pbest_pos[i] = particles[i].copy()

        gbest_idx = np.argmax(pbest_scores)
        if pbest_scores[gbest_idx] > gbest_score:
            gbest_score = pbest_scores[gbest_idx]
            gbest_pos = pbest_pos[gbest_idx].copy()

        print(f"Iteration {iteration+1}/{max_iter}, Best Score: {gbest_score:.2f}, Best Threshold: {int(gbest_pos[0])}")

    # Apply best threshold to binarize image
    threshold = int(gbest_pos[0])
    binary_img = (img_np > threshold) * 255

    # Display original and thresholded images
    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(img_np, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title(f"Thresholded Image (T={threshold})")
    plt.imshow(binary_img, cmap='gray')
    plt.axis('off')
    plt.show()
```

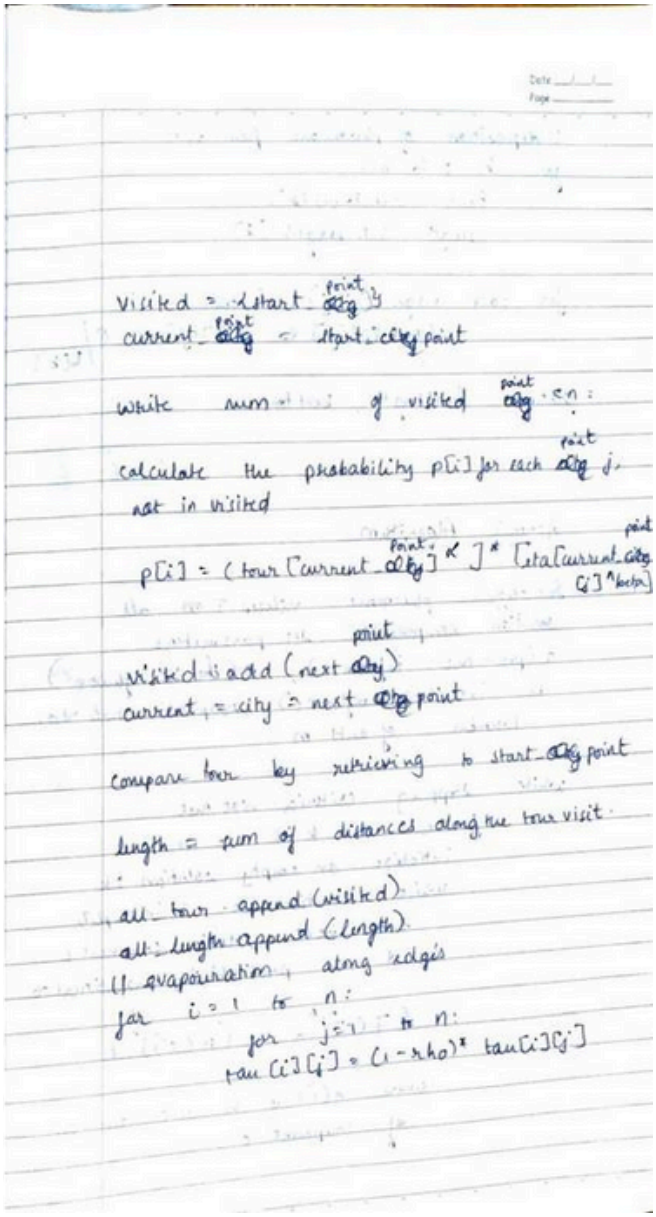
OUTPUT:

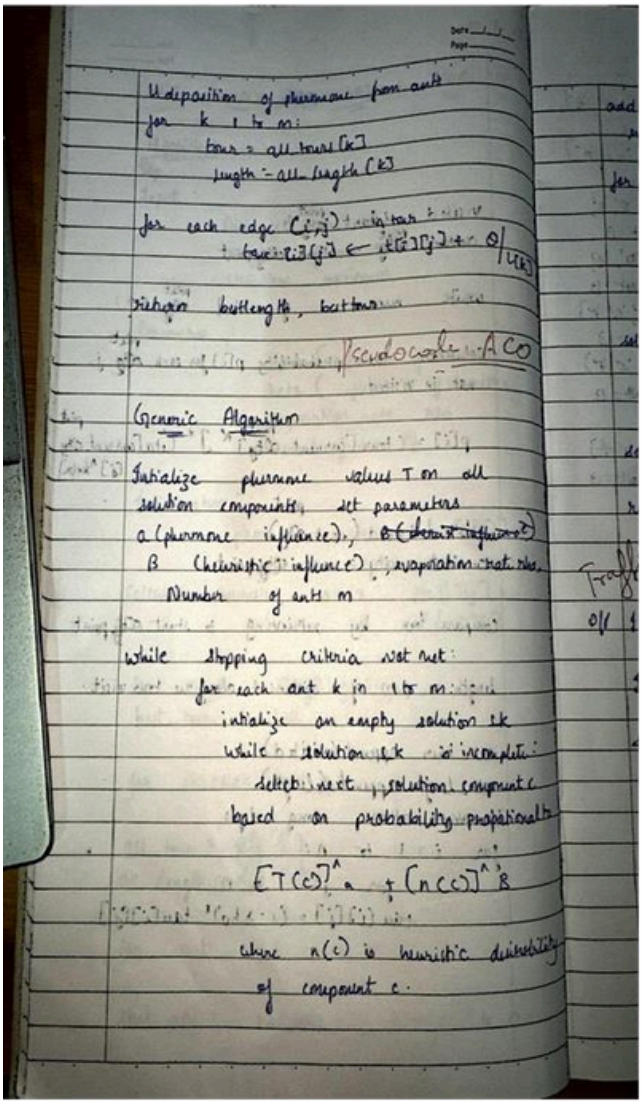
```
1bm23cs235-POTNURU ASHRITA
Gen 0 | best = 18.600 s | ['Pack', 'Pick', 'Place', 'Inspect', 'Label', 'Screw']
Gen 25 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 50 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 75 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 100 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 125 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 150 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 175 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 200 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 225 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Gen 250 | best = 18.400 s | ['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']

Best found sequence:
['Screw', 'Pick', 'Place', 'Inspect', 'Label', 'Pack']
Estimated total time: 18.400 s
```

Program 4: Ant colonization using travelling salesman problem

Algorithm:







Updation of pheromone from ants
for k = 1 to m:
tour = all_tours[k]
length = all_length[k]

for each edge (i,j) in tour:
tour[i][j] ← tour[i][j] + Q/length[k]

return best_length, best_tour

Pseudocode: ACO

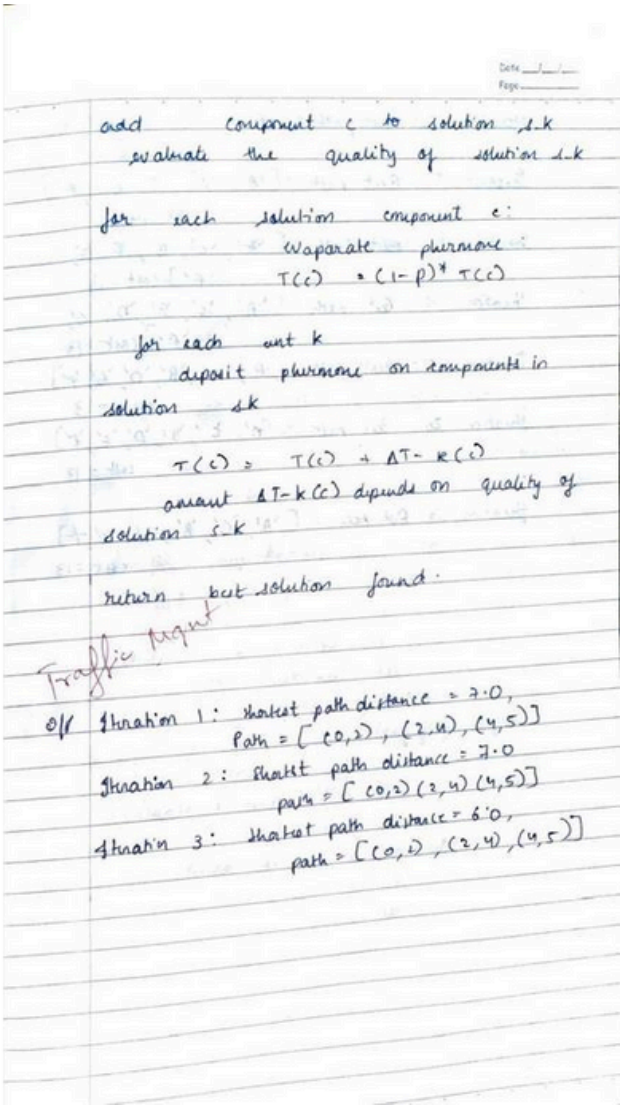
Genomic Algorithm

Initialize pheromone values T on all solution components, set parameters
α (pheromone influence), β (heuristic influence), evaporation rate, ρ
Number of ants m

while stopping criteria not met:
for each ant k in 1 to m:
initialize an empty solution sk
while solution sk is incomplete:
select next solution component c
based on probability proportional to

$$[T(c)]^{\alpha} * [n(c)]^{\beta}$$

where n(c) is heuristic desirability of component c.



add component c to solution s_k
evaluate the quality of solution s_k

for each solution component c:
Vaporate pheromone:
T(c) = (1-ρ) * T(c)

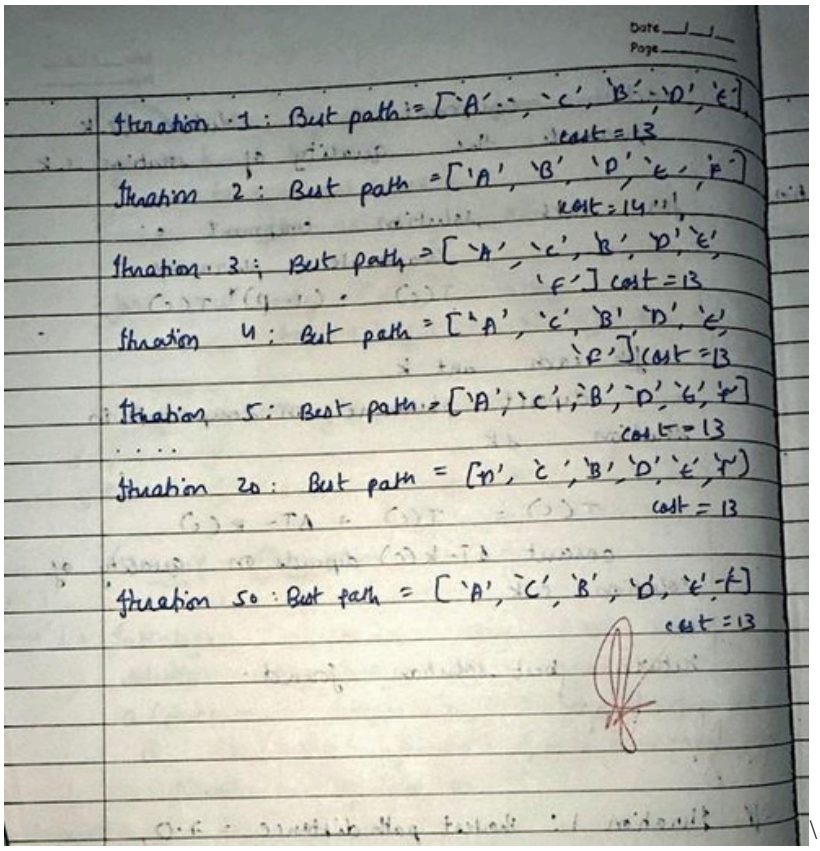
for each ant k
deposit pheromone on components in solution sk

T(c) = T(c) + ΔT-k(c)
amount ΔT-k(c) depends on quality of solution s_k

return best solution found.

Traffic light

Iteration 1: shortest path distance = 7.0,
Path = [(0,2), (2,4), (4,5)]
Iteration 2: Shortest path distance = 7.0
path = [(0,2) (2,4) (4,5)]
Iteration 3: Shortest path distance = 6.0,
path = [(0,2), (2,4), (4,5)]



CODE: -

```
import numpy as np

# Graph as adjacency matrix: 0 means no direct road
# Nodes: 0 - start, 4 - destination
graph = np.array([
[0, 2, 0, 1, 0],
[2, 0, 3, 2, 0],
[0, 3, 0, 0, 1],
[1, 2, 0, 0, 3],
[0, 0, 1, 3, 0]
])

num_nodes = graph.shape[0]
start_node = 0
end_node = 4

# Parameters
num_ants = 5
num_iterations = 10
alpha = 1.0 # pheromone influence
beta = 2.0 # heuristic influence (inverse of distance)
evaporation_rate = 0.5
pheromone_deposit = 1.0

# Initialize pheromone trails with a small positive number
pheromone = np.ones((num_nodes, num_nodes)) * 0.1

# Heuristic matrix (inverse of distance, avoid division by zero)
heuristic = np.zeros_like(graph, dtype=float)
foriinrange(num_nodes):

    forjin range(num_nodes):
```

```
        if graph[i, j] > 0:
            heuristic[i, j] = 1.0 / graph[i, j]

def choose_next_node(current, visited):
    probabilities = []
    neighbors = []
    for j in range(num_nodes):
        ifgraph[current,j]>0andjnotinvisited:
            tau=pheromone[current,j]**alpha
            eta = heuristic[current, j] ** beta
            probabilities.append(tau * eta)
            neighbors.append(j)
    probabilities = np.array(probabilities)
    if probabilities.sum() == 0:
        return None
    probabilities=probabilities/probabilities.sum()
    returnnp.random.choice(neighbors,p=probabilities)

def ant_walk():
    path = [start_node]
    visited = set(path)
    current = start_node
    while current != end_node:
        next_node=choose_next_node(current, visited)
        if next_node is None:
            # Dead end: restart path (or stop)
            return None
        path.append(next_node)
        visited.add(next_node)
        current = next_node
    return path

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

best_path = None
best_length = float('inf')
for iteration in range(num_iterations):
    all_paths = []
    for _ in range(num_ants):
        path = None
        while path is None:
            path = ant_walk()
        all_paths.append(path)

    # Evaporate pheromone
    pheromone *= (1 - evaporation_rate)
    #Depositpheromonebasedonpathquality
    for path in all_paths:
        length = path_length(path)
        if length < best_length:
        best_length = length
        best_path = path
        deposit_amount=pheromone_deposit / length
        for i in range(len(path)-1):
```

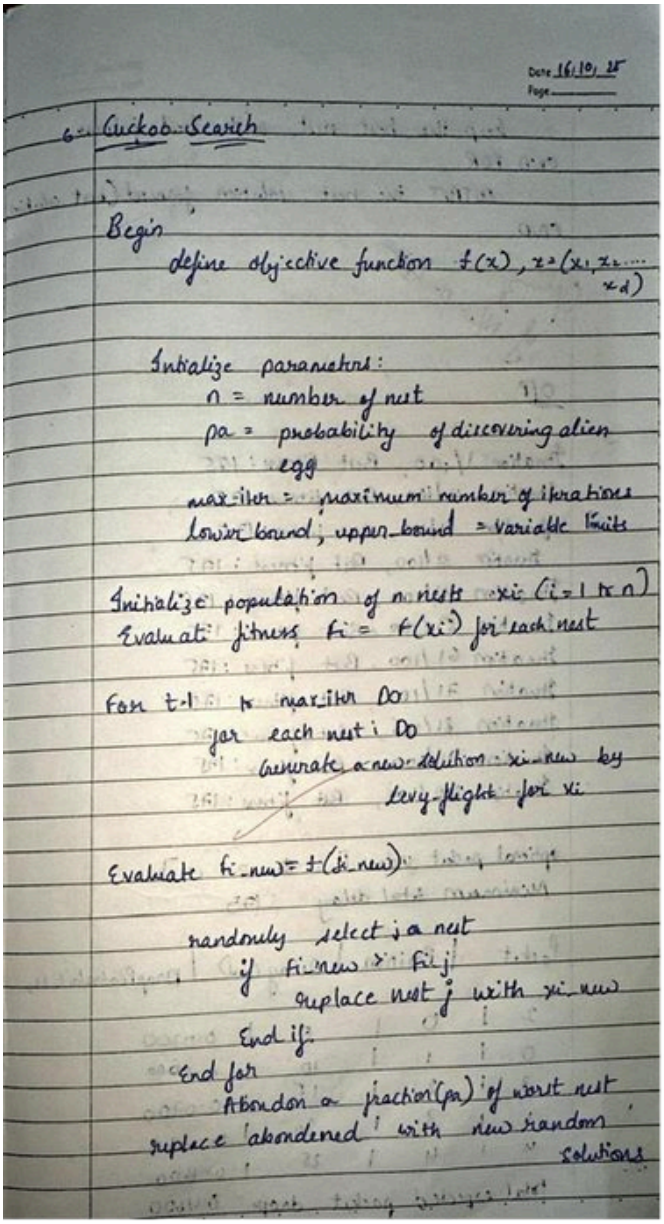
```
            pheromone[path[i], path[i+1]] += deposit_amount
            pheromone[path[i+1], path[i]] += deposit_amount # undirected graph

        print(f"Iteration {iteration+1}: Best path so far: {best_path} with length {best_length}")

    print("\nFinal best path found:", best_path)
    print("Path length:", best_length)
```

OUTPUT: -

```
1BM23CS235-POTNURU ASHRITA
Iteration 1: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 2: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 3: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 4: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 5: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 6: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 7: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 8: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 9: Best path so far: [0, np.int64(3), np.int64(4)] with length 4
Iteration 10: Best path so far: [0, np.int64(3), np.int64(4)] with length 4

Final best path found: [0, np.int64(3), np.int64(4)]
Path length: 4
```

Program 5: Cukoo Search using network packet transmission

Algorithm:



Date 16.10.25
Page

6. Cuckoo Search

Begin
define objective function f(x), x=(x1, x2, ... xd)

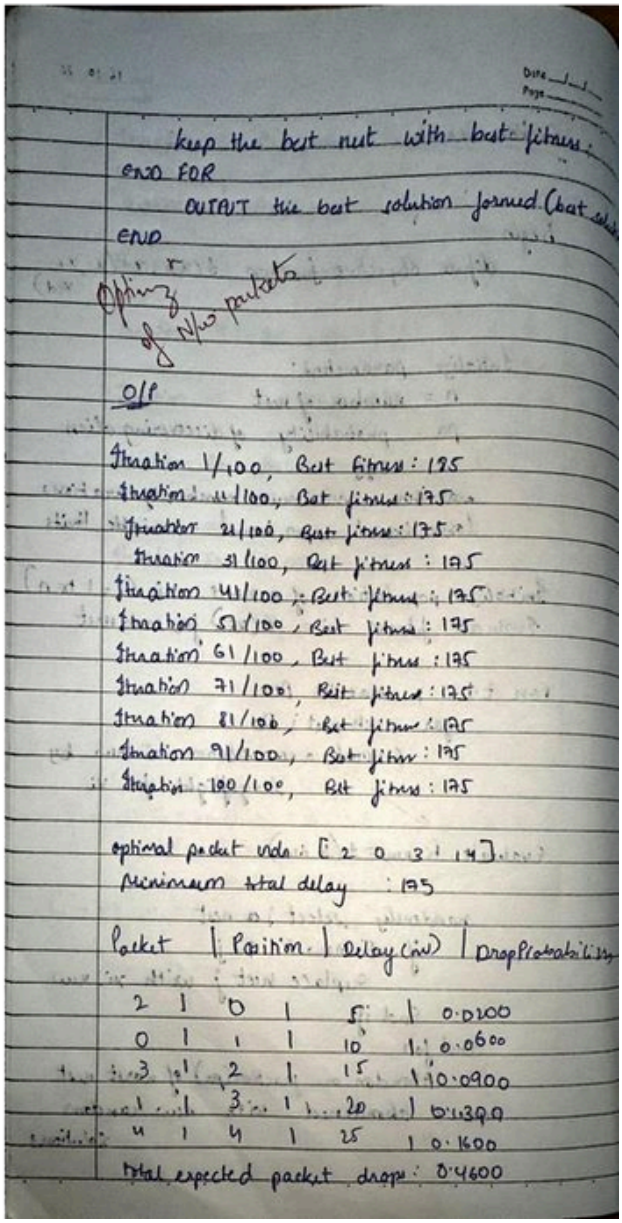
Initialize parameters:
n = number of nest
pa = probability of discovering alien egg
max_iter = maximum number of iterations
lower_bound, upper_bound = variable limits

Initialize population of n nests xi (i=1 to n)
Evaluate fitness fi = f(xi) for each nest

For t=1 to max_iter DO
for each nest i DO
Generate a new solution xi_new by Levy-flight for xi

Evaluate fi_new = f(xi_new)

randomly select j a nest
if fi_new > fij
replace nest j with xi_new
End if
End for
Abandon a fraction(pa) of worst nest
replace abandened with new random solutions



keep the best nest with best fitness
END FOR
OUTPUT the best solution formed (best

END

Optimize of N/w packets

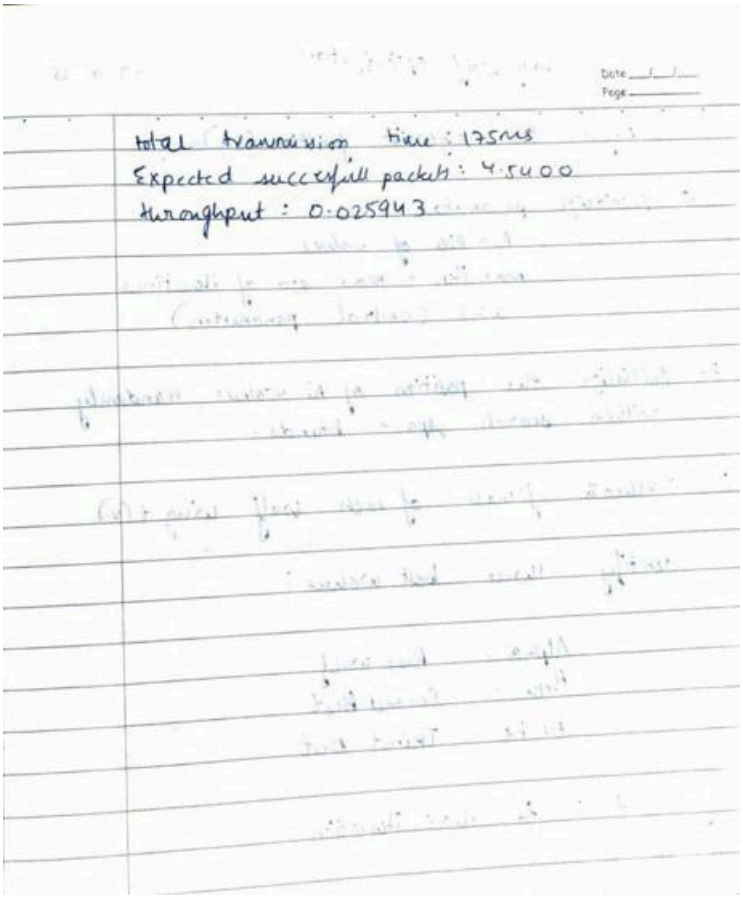
O/P

Iteration 1/100, Best fitness: 175
Iteration 11/100, Best fitness: 175
Iteration 21/100, Best fitness: 175
Iteration 31/100, Best fitness: 175
Iteration 41/100, Best fitness: 175
Iteration 51/100, Best fitness: 175
Iteration 61/100, Best fitness: 175
Iteration 71/100, Best fitness: 175
Iteration 81/100, Best fitness: 175
Iteration 91/100, Best fitness: 175
Iteration 100/100, Best fitness: 175

optimal packet order: [2 0 3 1 4]
Minimum total delay: 175

Packet	Position	Delay (ms)	DropProbability
2	1	5	0.0200
0	1	10	0.0600
3	2	15	0.0900
1	3	20	0.1300
4	4	25	0.1600

Total expected packet drops: 0.4600



CODE: -

```
importnumpy as np

#Example packet delays (in ms)
packet_delays = [10, 20, 5, 15, 25]
#Objective function: minimize total completion time for given packet order
defobjective(order):

    total_time = 0
    current_time = 0
    foriin order:
        current_time += packet_delays[i]
        total_time += current_time
    return total_time

#Generate a random solution (permutation of packets)
defrandom_solution(n):
    solution = np.arange(n)
    np.random.shuffle(solution)
    return solution

#Levyflight step (for continuous problems, adapted for permutation with swap)
deflevy_flight(solution):
    n=len(solution)
    #Swap two random positions
    i,j=np.random.choice(n, 2, replace=False)
    new_solution = solution.copy()
    new_solution[i], new_solution[j] = new_solution[j], new_solution[i]
    return new_solution
```

```python
def cuckoo_search(n_packets, n_nests=15, n_iterations=100, pa=0.25):
    # Initialize nests with random solutions
    nests = [random_solution(n_packets) for _ in range(n_nests)]
    fitness = np.array([objective(nest) for nest in nests])

    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx].copy()
    best_fitness = fitness[best_idx]

    for iter in range(n_iterations):

        # Generate new solutions by Levy flights
        for i in range(n_nests):
            new_nest = levy_flight(nests[i])
            new_fitness = objective(new_nest)

            # Greedy selection: keep better solutions
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness
                # Update global best
                if new_fitness < best_fitness:
                    best_fitness = new_fitness
                    best_nest = new_nest.copy()

        # Abandon some nests and generate new ones
        for i in range(n_nests):
            if np.random.rand() < pa:
                nests[i] = random_solution(n_packets)
                fitness[i] = objective(nests[i])

                # Update global best
                if fitness[i] < best_fitness:
                    best_fitness = fitness[i]
                    best_nest = nests[i].copy()

        if iter % 10 == 0 or iter == n_iterations - 1:
            print(f"Iteration {iter+1}/{n_iterations}, Best fitness: {best_fitness}")

    return best_nest, best_fitness

# Run the Cuckoo Search optimization
best_order, best_cost = cuckoo_search(len(packet_delays))
print("\nOptimal packet order (indices):", best_order)
print("Minimum total delay:", best_cost)
```

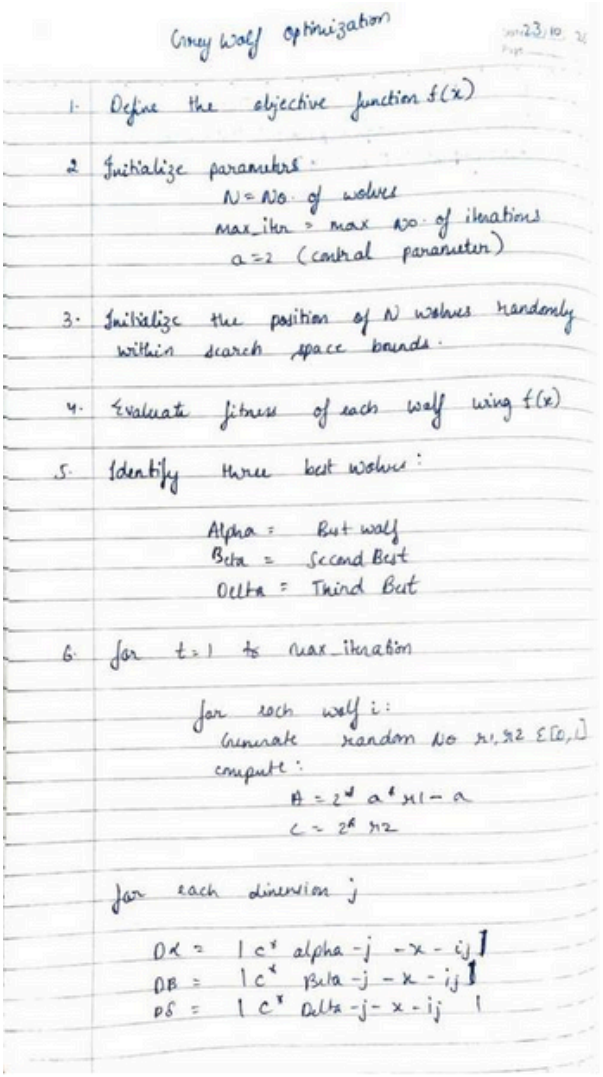
OUTPUT: -

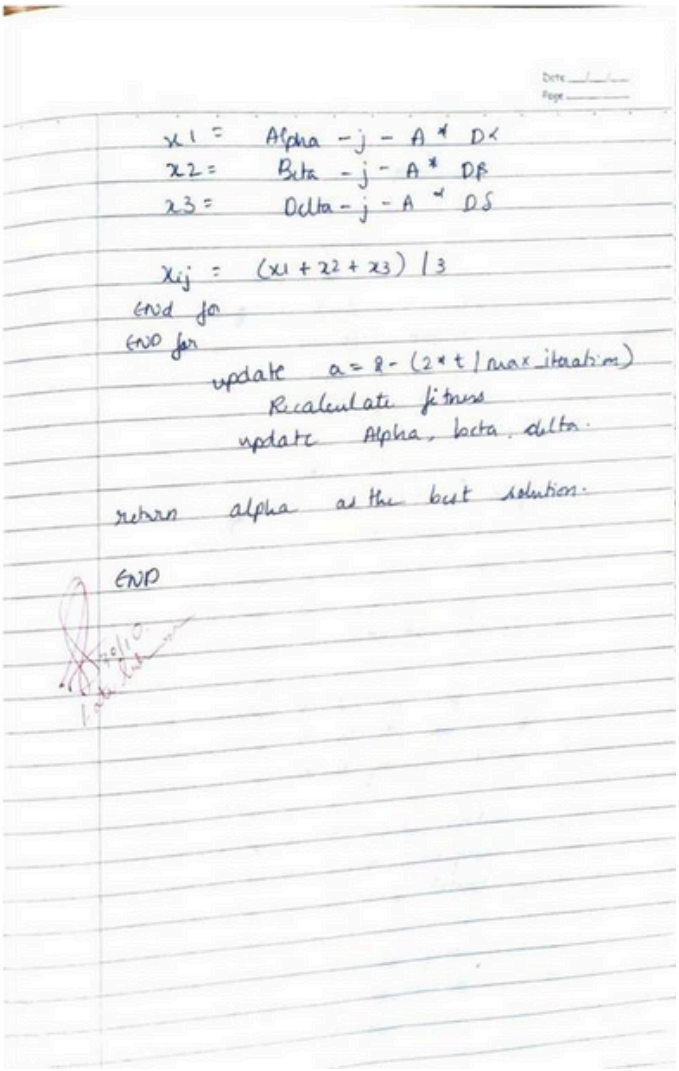
```
1BM23CS235-POTNURU ASHRITA
Iteration 1/100, Best fitness: 185
Iteration 11/100, Best fitness: 175
Iteration 21/100, Best fitness: 175
Iteration 31/100, Best fitness: 175
Iteration 41/100, Best fitness: 175
Iteration 51/100, Best fitness: 175
Iteration 61/100, Best fitness: 175
Iteration 71/100, Best fitness: 175
Iteration 81/100, Best fitness: 175
Iteration 91/100, Best fitness: 175
Iteration 100/100, Best fitness: 175

Optimal packet order (indices): [2 0 3 1 4]
Minimum total delay: 175
```

[ ]:

Program 6: Grey Wolf Optimization

Algorithm:





CODE: -

```
# gwo_reliability.py

# Grey Wolf Optimization (GWO) for Reliability-Redundancy Allocation (series system)

# Python 3.9+

from future import annotations

import math

import random

from dataclasses import dataclass
```

```
        from typing import List, Optional, Tuple

        class RRAData:

        p: List[float] # component reliability for subsystem i (0)

    c: List[float] # cost per component in subsystem i

        w: Optional[List[float]] = None # weight per component (optional)

        max_red: Optional[List[int]] = None # max redundancy per subsystem (>=1)

        budget: Optional[float] = None

        weight_limit: Optional[float] = None

        def post_init (self):

        m = len(self.p)

        assert len(self.c) == m

        if self.w is not None:

        assert len(self.w) == m

        if self.max_red is None:

        self.max_red = [5] * m # default caps

        else:

        assert len(self.max_red) == m

        def subsystem_reliability(p: float, x: int) -> float:

        # Reliability of x identical parallel components

        x = max(1, x)

        return 1.0 - (1.0 - p) ** x

        def system_reliability(x: List[int], data: RRAData) -> float:

        R = 1.0

        for xi, pi in zip(x, data.p):

        Ri = subsystem_reliability(pi, xi)

        R *= Ri

        if R <= 0.0:

        return 0.0

        return R

        def totals(x: List[int], data: RRAData) -> Tuple[float, float]:
```

```
return tot_cost, tot_weight

def penalty(x: List[int], data: RRAData) -> float:

tot_cost, tot_weight = totals(x, data)

v_cost = max(0.0, (tot_cost - (data.budget if data.budget is not None else float('inf'))))

v_weight = 0.0

if data.weight_limit is not None and data.w is not None:

v_weight = max(0.0, tot_weight - data.weight_limit)

# Scale penalties relative to typical magnitudes

# Add small eps to avoid division by zero if limits are None

scale_c = (data.budget if data.budget is not None else (tot_cost + 1.0))

scale_w = (data.weight_limit if data.weight_limit is not None else (tot_weight + 1.0))

return 1e3 * (v_cost / (scale_c + 1e-9))**2 + 1e3 * (v_weight / (scale_w + 1e-9))**2

def objective(x_cont: List[float], data: RRAData) -> float:

x = []

for val, xmax in zip(x_cont, data.max_red):

xi = int(round(val))

xi = max(1, min(xi, xmax))

x.append(xi)

R = system_reliability(x, data)

pen = penalty(x, data)

return -(R) + pen # minimize

def clamp_position(pos: List[float], data: RRAData) -> List[float]:

out = []

for v, xmax in zip(pos, data.max_red):

out.append(min(max(v, 1.0), float(xmax)))

return out

class GreyWolfOptimizer:

def init (self,

data: RRAData,
```

```
n_wolves: int = 30,

n_iters: int = 300,

a0: float = 2.0,

seed: Optional[int] = 42):

self.data = data

self.m = len(data.p)

self.n_wolves = n_wolves

self.n_iters = n_iters

self.a0 = a0

if seed is not None:

random.seed(seed)

self.pop: List[List[float]] = [

[random.uniform(1.0, float(xmax)) for xmax in data.max_red] for _ in range(n_wolves)

]

self.fvals: List[float] = [objective(x, data) for x in self.pop]

self.alpha = None # type: Optional[List[float]]

self.beta = None

self.delta = None

self.alpha_val = float('inf')

self.beta_val = float('inf')

self.delta_val = float('inf')

self._update_leaders()

def _update_leaders(self):

ranked = sorted(zip(self.fvals, self.pop), key=lambda t: t[0])

self.alpha_val, self.alpha = ranked[0]

self.beta_val, self.beta = ranked[1]

self.delta_val, self.delta = ranked[2]

def step(self, t: int):

a = self.a0 * (1 - t / max(1, self.n_iters - 1))
```

```
new_pop: List[List[float]] = []

new_f: List[float] = []

for i in range(self.n_wolves):

X = self.pop[i]

X1 = []

X2 = []

X3 = []

for d in range(self.m):

r1a, r2a = random.random(), random.random()

r1b, r2b = random.random(), random.random()

r1c, r2c = random.random(), random.random()

A1 = 2 * a * r1a - a

C1 = 2 * r2a

A2 = 2 * a * r1b - a

C2 = 2 * r2b

A3 = 2 * a * r1c - a

C3 = 2 * r2c

D_alpha = abs(C1 * self.alpha[d] - X[d])

D_beta = abs(C2 * self.beta[d] - X[d])

D_delta = abs(C3 * self.delta[d] - X[d])

X1d = self.alpha[d] - A1 * D_alpha

X2d = self.beta[d] - A2 * D_beta

X3d = self.delta[d] - A3 * D_delta

X1.append(X1d); X2.append(X2d); X3.append(X3d)

X_new = [(x1 + x2 + x3) / 3.0 for x1, x2, x3 in zip(X1, X2, X3)]

X_new = clamp_position(X_new, self.data)

f_new = objective(X_new, self.data)

new_pop.append(X_new)

new_f.append(f_new)
```

```
self.pop = new_pop

self.fvals = new_f

self._update_leaders()

def run(self, verbose: bool = True) -> Tuple[List[int], float, dict]:

history = []

for t in range(self.n_iters):

self.step(t)

if verbose and (t % max(1, self.n_iters // 10) == 0 or t == self.n_iters - 1):

print(f"[iter {t+1:4d}] best (−R+pen)={self.alpha_val:.6f}")

history.append(self.alpha_val)

best_cont = self.alpha

best_x = [int(round(v)) for v in best_cont]

# clamp to bounds

best_x = [max(1, min(xi, xmax)) for xi, xmax in zip(best_x, self.data.max_red)]

best_R = system_reliability(best_x, self.data)

feas_pen = penalty(best_x, self.data)

return best_x, best_R, {

"objective": -best_R + feas_pen,

"cost_weight": totals(best_x, self.data),

"history": history

}

if name == " main ":

data = RRAData(

p=[0.90, 0.92, 0.88, 0.93, 0.91],

c=[12.0, 10.0, 9.0, 11.0, 8.0],

w=[1.6, 1.2, 1.4, 1.5, 1.0], # optional weight per component

max_red=[6, 6, 6, 6, 6], # cap per subsystem

budget=120.0, # total cost constraint (optional)

weight_limit=18.0 # total weight constraint (optional)
```

```
)
gwo = GreyWolfOptimizer(
data=data,
n_wolves=35,
n_iters=400,
a0=2.0,
seed=123
)
best_x, best_R, info = gwo.run(verbose=True)
tot_cost, tot_weight = info["cost_weight"]
print("\n=== GWO Result (Reliability Optimization) ===")
print("Best redundancy vector x (per subsystem):", best_x)
print("System reliability R(x):", f"{best_R:.8f}")
print("Total cost:", f"{tot_cost:.2f}")
print("Total weight:", f"{tot_weight:.2f}")
print("Feasible (budget/weight):",
("Yes" if (data.budget is None or tot_cost <= data.budget) and
(data.weight_limit is None or tot_weight <= data.weight_limit) else "No"))
```

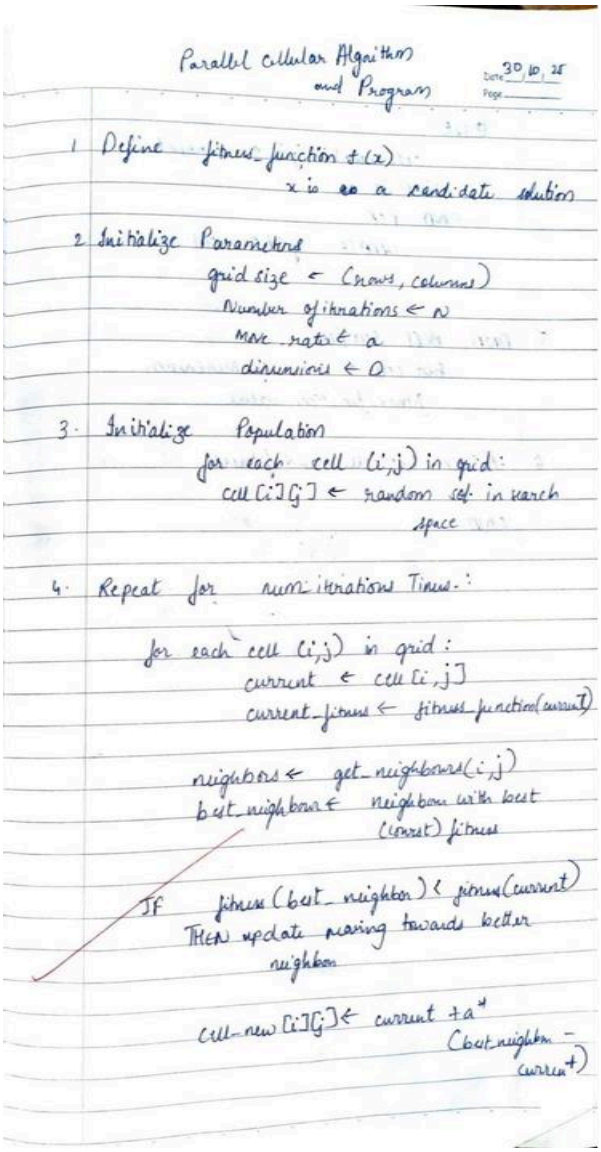
OUTPUT:-

Output Clear

```
[iter    1] best (-R+pen)=-0.963059
[iter   41] best (-R+pen)=-0.920857
[iter   81] best (-R+pen)=-0.974981
[iter  121] best (-R+pen)=-0.974981
[iter  161] best (-R+pen)=-0.976440
[iter  201] best (-R+pen)=-0.976440
[iter  241] best (-R+pen)=-0.976440
[iter  281] best (-R+pen)=-0.976440
[iter  321] best (-R+pen)=-0.976440
[iter  361] best (-R+pen)=-0.976440
[iter  400] best (-R+pen)=-0.976440

=== GWO Result (Reliability Optimization) ===
Best redundancy vector x (per subsystem): [2, 2, 3, 2, 3]
System reliability R(x): 0.97644026
Total cost: 117.00
Total weight: 15.80
Feasible (budget/weight): Yes
```

Program 7: Parallel Cellular Algorithms And Program by suing noise reduction image porcessing

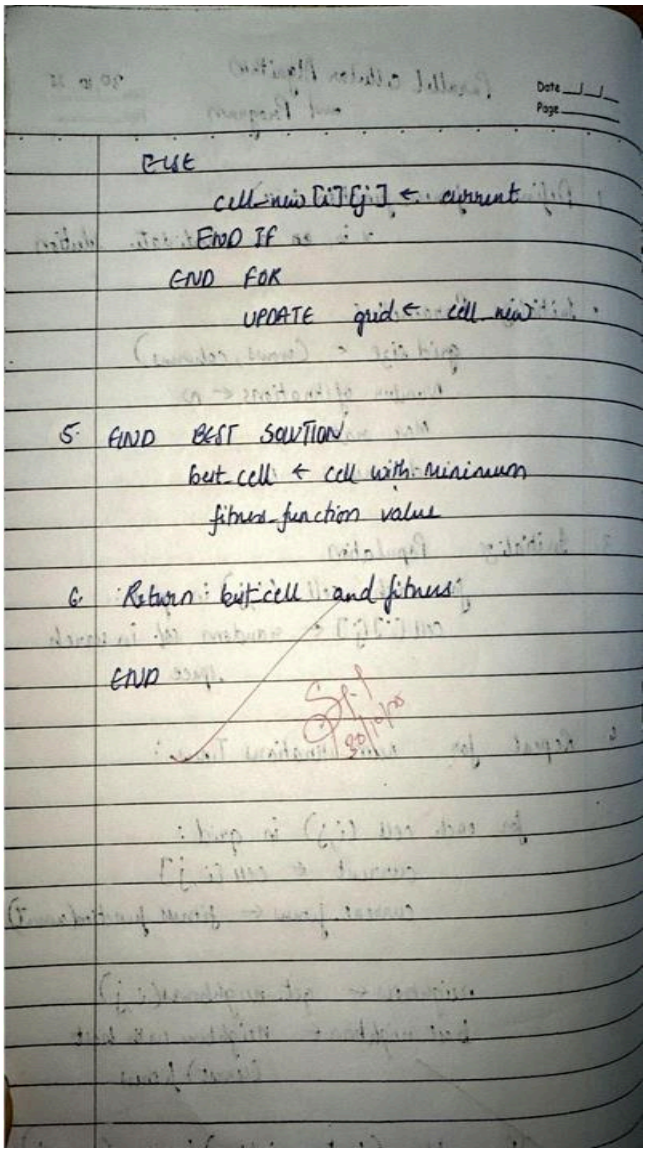
Algorithm:-



Parallel cellular Algorithm and Program

Date 30/10/25

1. Define fitness_function f(x)
   x is a candidate solution
2. Initialize Parameters
   grid size ← (rows, columns)
   Number of iterations ← n
   Move rate ← a
   dimensions ← D
3. Initialize Population
   for each cell (i,j) in grid:
   cell[i][j] ← random set in search space
4. Repeat for num_iterations Times:
   for each cell (i,j) in grid:
   current ← cell[i,j]
   current_fitness ← fitness_function(current)
   neighbors ← get_neighbors(i,j)
   best_neighbor ← neighbor with best (lowest) fitness
   IF fitness(best_neighbor) < fitness(current)
   THEN update moving towards better neighbor
   cell_new[i][j] ← current + a* (best_neighbor - current)



ELSE
cell_new[i][j] ← current
END IF
END FOR
UPDATE grid ← cell_new
5. FIND BEST SOLUTION
best_cell ← cell with minimum fitness_function value
6. Return best_cell and fitness
END

Image Processing

Initial Image:

20	30	40	30	20
30	100	150	100	30
40	150	255	150	40
30	100	150	100	30
20	30	40	30	20

Iteration 1:

30	42	54	42	30
42	96	134	96	42
54	134	208	134	54
42	96	134	96	42
30	42	54	42	30

Iteration 2:

39	51	63	51	39
51	92	122	92	51
63	122	174	122	63
51	92	122	92	51
39	51	63	51	39

.....

Iteration 5:

57	67	75	67	57
67	85	98	85	67
75	98	119	98	75
67	85	98	85	67
57	67	75	67	57

Key Improvements in Research Paper:

→ Neighbourhood:

Early CAs often used very simple neighbourhoods (eg 4 to 9 neighbour)

Now CAs works on larger and adaptive neighbour (25 neighbour models)

→ Scalability to larger grids.

→ Recent works/techniques made the number of iterations to be reduced

So shorter runtime

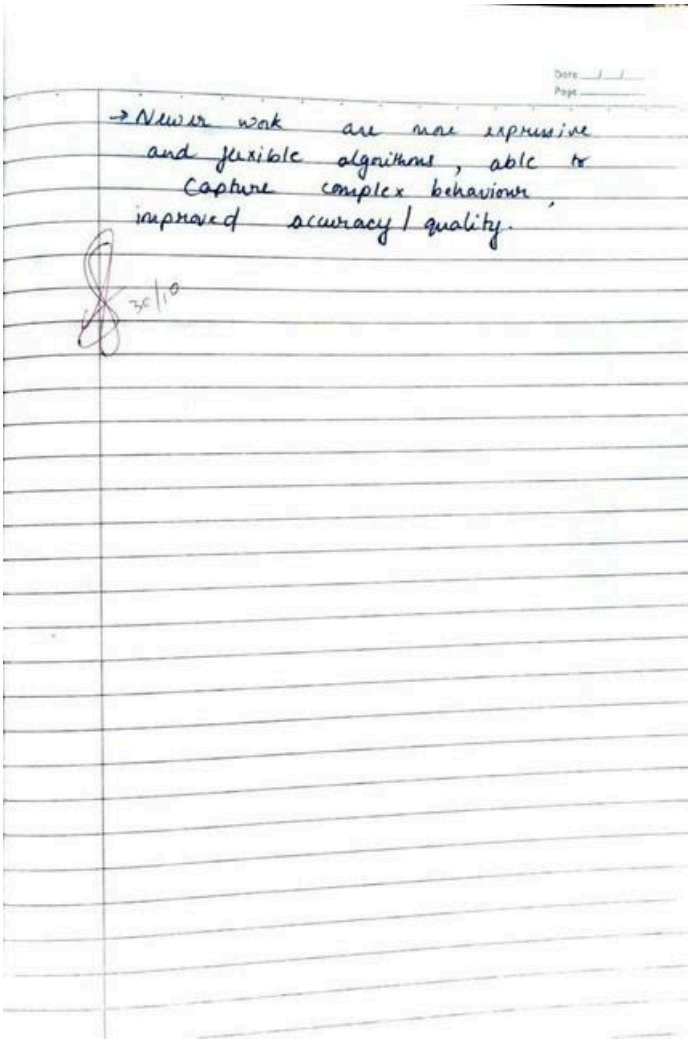
→ Better parallel implementation techniques

Old algorithms suffered from synchronization overheads, insufficient memory access, or poor use of data locality.

Benefit: More efficient per cell update, better scaling, less overhead.

→ Old CA's focused on shared-memory or single-machine parallelism.

→ Recent CA's work is distributed/cloud/ Internet-scale frameworks



CODE: -

```
# ---SIMPLEPARALLEL CELLULAR ALGORITHM (NO IMPORTS) ---
# Example grayscale "image" (each pixel: 0–255)
image = [


    [ 20, 30, 40, 30, 20], [ 30,
    100, 150, 100, 30], [ 40,
    150, 255, 150, 40], [ 30,
    100, 150, 100, 30], [ 20,
    30, 40, 30, 20]
]
# Parameters
iterations = 5
alpha = 0.4         # number of iterations
                    # blending factor (how much neighbor average influences pixel)
height = len(image)
width = len(image[0])
defget_neighbors(img, x, y):

    """Return 3x3 neighborhood around (x,y) with edge handling."""
    neighbors = []
    for i in range(x-1, x+2):
```

```
        for j in range(y-1, y+2):
            if 0 <= i < height and 0 <= j < width:
                neighbors.append(img[i][j])
    return neighbors

def parallel_update(img):
    """Perform one parallel update step."""
    new_img = [[0]*width for _ in range(height)]
    for i in range(height):
        for j in range(width):
            neighbors = get_neighbors(img, i, j)
            mean_value = sum(neighbors) / len(neighbors)
            new_img[i][j] = int((1 - alpha) * img[i][j] + alpha * mean_value)
    return new_img

def show_image(img):
    """Print the image matrix (simulated grayscale view)."""
    for row in img:
        print(" ".join(f"{v:3}" for v in row))
    print("-" * (width * 4))

# --- Run iterations and show results ---
print("Initial Image:")
show_image(image)

fortin range(1, iterations + 1):
    image = parallel_update(image)
    print(f"Iteration {t}:")
    show_image(image)
```

OUTPUT: -

```
P.Ashrita 1BM23CS235
Initial Image:
 20  30  40  30  20
 30 100 150 100  30
 40 150 255 150  40
 30 100 150 100  30
 20  30  40  30  20
--------------------
Iteration 1:
 30  42  54  42  30
 42  96 134  96  42
 54 134 208 134  54
 42  96 134  96  42
 30  42  54  42  30
--------------------
Iteration 2:
 39  51  63  51  39
 51  92 122  92  51
 63 122 174 122  63
 51  92 122  92  51
 39  51  63  51  39
--------------------
Iteration 3:
 46  58  69  58  46
 58  89 112  89  58
 69 112 150 112  69
 58  89 112  89  58
 46  58  69  58  46
--------------------
Iteration 4:
 52  63  73  63  52
 63  87 104  87  63
 73 104 132 104  73
 63  87 104  87  63
 52  63  73  63  52
--------------------
Iteration 5:
 57  67  75  67  57
 67  85  98  85  67
 75  98 119  98  75
 67  85  98  85  67
 57  67  75  67  57
--------------------
```