VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Hitesh Sharma(1BM23CS114)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug-2025 to Dec-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Hitesh Sharma (1BM23CS114),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

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

Index

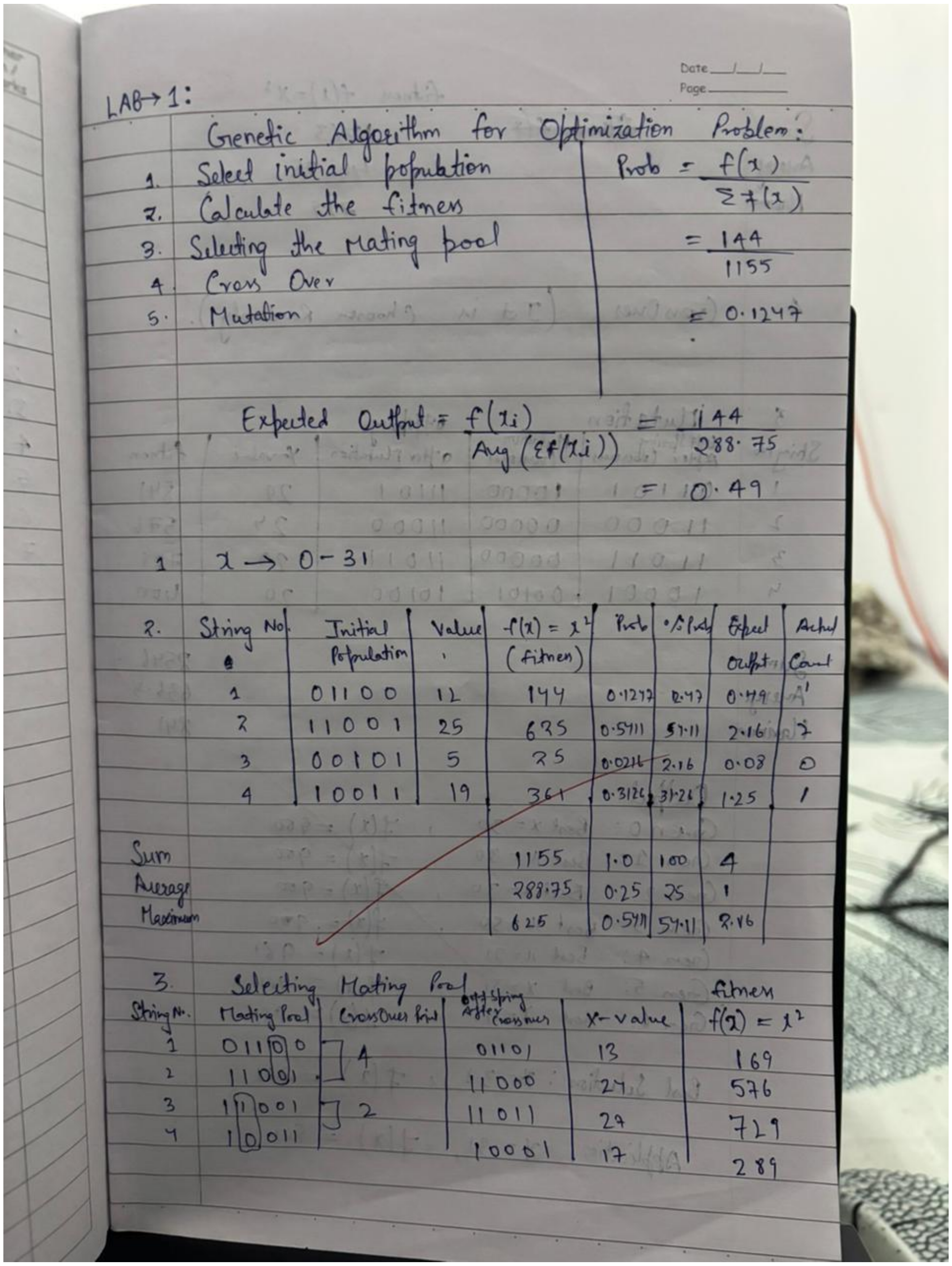
Github Link:

https://github.com/Hitesh6747/BIS_LAB

Program 1

We have multiple jobs and limited resources and we need to assign them to minimize completion time, cost or maximize efficiency.

ALGORITHM:



Date __/__/__
Page ______

LAB→1:

Genetic Algorithm for Optimization Problem:

1. Select initial population
2. Calculate the fitness
3. Selecting the mating pool
4. Cross Over
5. Mutation

$$\text{Prob} = \frac{f(x)}{\Sigma f(x)} = \frac{144}{1155} = 0.1247$$

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

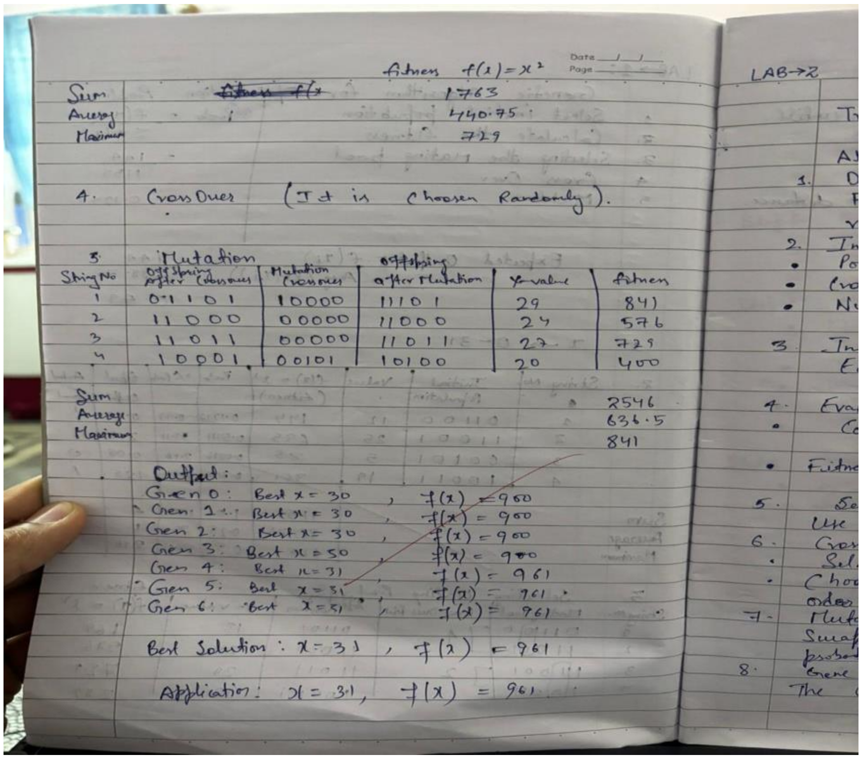
1. $x \to 0 - 31$

2.

String No.	Initial Population	Value	$f(x) = x^2$ (fitness)	Prob	% Prob	Expected Output	Actual Count
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.16	2
3	00101	5	25	0.0216	2.16	0.08	0
4	10011	19	361	0.3126	31.26	1.25	1
Sum			1155	1.0	100	4	
Average			288.75	0.25	25	1	
Maximum			625	0.5411	54.11	2.16	

3. Selecting Mating Pool

String No.	Mating Pool	Cross Over Point	Offspring After Crossover	x-value	fitness $f(x) = x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289



fitness $f(x) = x^2$

Sum	~~fitness f(x~~	1763
Average		440.75
Maximum		729

4\. Cross Over (It is Choosen Randomly).

5\. Mutation

String No	Offspring After Crossover	Mutation Crossover	Offspring after Mutation	x-value	fitness
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
Sum					2546
Average					636.5
Maximum					841

Output:
Gen 0: Best x = 30 , f(x) = 900
Gen 1: Best x = 30 , f(x) = 900
Gen 2: Best x = 30 , f(x) = 900
Gen 3: Best x = 50 , f(x) = 900
Gen 4: Best x = 31 , f(x) = 961
Gen 5: Best x = 31 , f(x) = 961
Gen 6: Best x = 31 , f(x) = 961

Best Solution : x = 31 , f(x) = 961

Application: x = 31, f(x) = 961

Code:

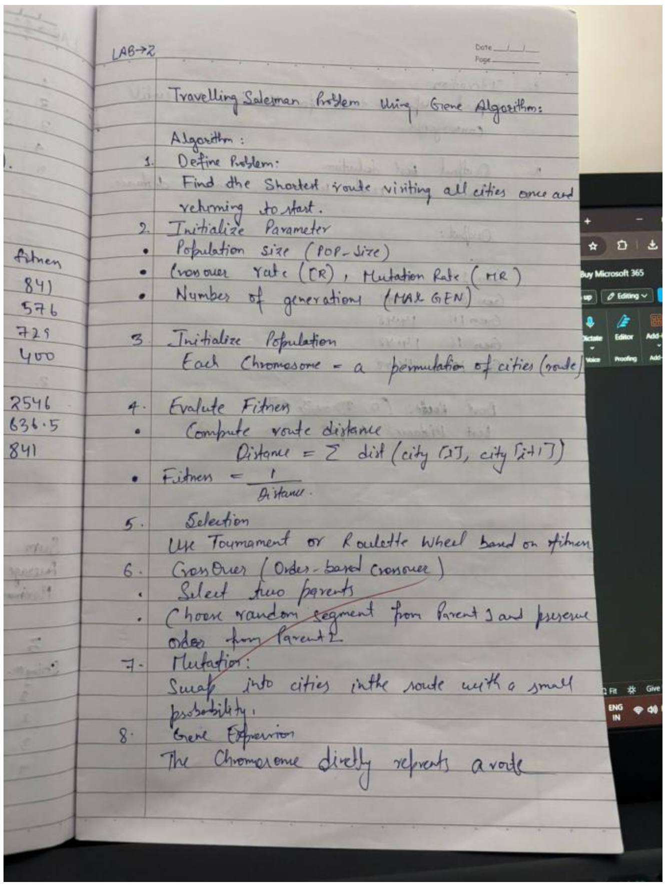
```python
import numpy as np

target_position = np.array([10.0, 10.0])
num_particles = 30
num_iterations = 50
w = 0.7
c1 = 1.5
c2 = 1.5
v_max = 1.0

positions = np.random.uniform(0, 10, (num_particles, 2))
velocities = np.random.uniform(-1, 1, (num_particles, 2))
personal_best_positions = positions.copy()
personal_best_values = np.full(num_particles, np.inf)
global_best_position = np.zeros(2)
global_best_value = np.inf

def fitness(position, velocity):
```

```python
    distance = np.linalg.norm(position - target_position)
    smoothness_penalty = np.linalg.norm(velocity)
    return distance + 0.3 * smoothness_penalty

for t in range(num_iterations):
    for i in range(num_particles):
        fit = fitness(positions[i], velocities[i])
        if fit < personal_best_values[i]:
            personal_best_values[i] = fit
            personal_best_positions[i] = positions[i].copy()
        if fit < global_best_value:
            global_best_value = fit
            global_best_position = positions[i].copy()
    for i in range(num_particles):
        r1, r2 = np.random.rand(2)
        velocities[i] = (
            w * velocities[i]
            + c1 * r1 * (personal_best_positions[i] - positions[i])
            + c2 * r2 * (global_best_position - positions[i])
        )
        velocities[i] = np.clip(velocities[i], -v_max, v_max)
        positions[i] += velocities[i]

print("Final optimized position:", global_best_position)
print("Minimum fitness value:", global_best_value)
```

Program 2

The Travelling Salesman Problem (TSP) asks for the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The provided text describes using a Genetic Algorithm to solve this by evolving city sequences (chromosomes) through selection, crossover, and mutation to minimize the total tour distance.

Algorithm:


LAB→2

Travelling Salesman Problem Using Gene Algorithm:

Algorithm :

1. Define Problem:
   Find the Shortest route visiting all cities once and returning to start.
2. Initialize Parameter
   - Population size (POP-Size)
   - Cross over rate (CR), Mutation Rate (MR)
   - Number of generations (MAX GEN)
3. Initialize Population
   Each Chromosome = a permutation of cities (route)
4. Evalute Fitness
   - Compute route distance
     $$Distance = \sum dist(city[i], city[i+1])$$
   - $Fitness = \frac{1}{Distance}$.
5. Selection
   Use Tournament or Roulette Wheel based on fitness
6. Cross Over (Order-based crossover)
   - Select two parents
   - Choose random segment from Parent 1 and preserve order from Parent 2
7. Mutation:
   Swap into cities in the route with a small probability.
8. Gene Expression
   The Chromosome directly represents a route

Fitness
841
576
729
400

2546
636.5
841

9. Iteration
- Repeat steps 4 to 8 for Max Gen or until convergence.

10. Output Best Solution
- Track the route with the minimum distance.

Output:

Gen 1: 18.30
Gen 6: 15.92
Gen 11: 14.48
Gen 16: 14.48
Gen 20: 14.48

Best Route: [0, 1, 2, 3, 4]
Best Distance: 14.48

Code:

```
import random
import math

cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]
num_cities = len(cities)
population_size = 30
generations = 200
crossover_rate = 0.8
mutation_rate = 0.2

def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def tour_length(chromosome):
    length = 0
```

```
    for i in range(num_cities):
        length += distance(cities[chromosome[i]],
cities[chromosome[(i+1)%num_cities]])
    return length

def fitness(chromosome):
    return 1 / tour_length(chromosome)

def initial_population():
    population = []
    for _ in range(population_size):
        chromosome = list(range(num_cities))
        random.shuffle(chromosome)
        population.append(chromosome)
    return population

def selection(population):
    contenders = random.sample(population, 3)
    contenders.sort(key=lambda c: fitness(c), reverse=True)
    return contenders[0]

def crossover(p1, p2):
    if random.random() < crossover_rate:
        a, b = sorted(random.sample(range(num_cities), 2))
        child = [-1]*num_cities
        child[a:b] = p1[a:b]
        fill = [x for x in p2 if x not in child]
        j = 0
        for i in range(num_cities):
            if child[i] == -1:
                child[i] = fill[j]
                j += 1
        return child
    return p1[:]

def mutate(chromosome):
    if random.random() < mutation_rate:
        a, b = random.sample(range(num_cities), 2)
        chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
    return chromosome

population = initial_population()
best_solution = None
best_distance = float("inf")
```

```python
for g in range(generations):
    new_pop = []
    for _ in range(population_size):
        parent1 = selection(population)
        parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        new_pop.append(child)
    population = new_pop
    for chromo in population:
        d = tour_length(chromo)
        if d < best_distance:
            best_distance = d
            best_solution = chromo

print("Best Tour (order of cities):", best_solution)
print("Best Tour Distance:", best_distance)
```

Program 3

Particle Swarm Optimization for smooth character movement using postion and velociy

Algorithm:

Lab→3

Particle Swarm Optimization For Smooth Character Movement using Position and Velocity.

Algorithm:

Step 1: Define the Problem

State variables: Position of the character in space

Velocity vector to be optimized

Step 2: Initialize PSO Parameters

- Number of Particles N
- Number of iteration T
- Inertia weight w
- Cognitive coefficient c_1
- Social coefficient c_2
- Velocity and position bounds (e.g. velocity limit)

Step 3. Initialize Particles

- For each particle:

Simulate character movement by updating position using the particle velocity.

New position = current position + velocity × Δt

- Compute Fitness value:

fitness = distance (new position, target position + Smoothness

- Smoothness penalty can be the magnitude of velocity changes.

Step5: Update Personal and Global Bests

Step6: Update Velocity and Position for each Particle

$$v_i^{t+1} = w \cdot v_i^t + c_1 \cdot r_1 (pbest_i - x_i^t) + c_2 r_2 (gbest - x_i^t)$$

Update particle position

$$x_i^{t+1} = x_i^t + v_i^{t+1}$$

Step7: Iterate

Step8: Output:

Output:

Start : [0,0]

Goal : [10,10]

Best velocity vector found after 20 iteration: [9.6589777, 9.77723]

Expected new position after dt = 1s:

Distance to target: 0.3571

Fitness : 1.3218

Code:

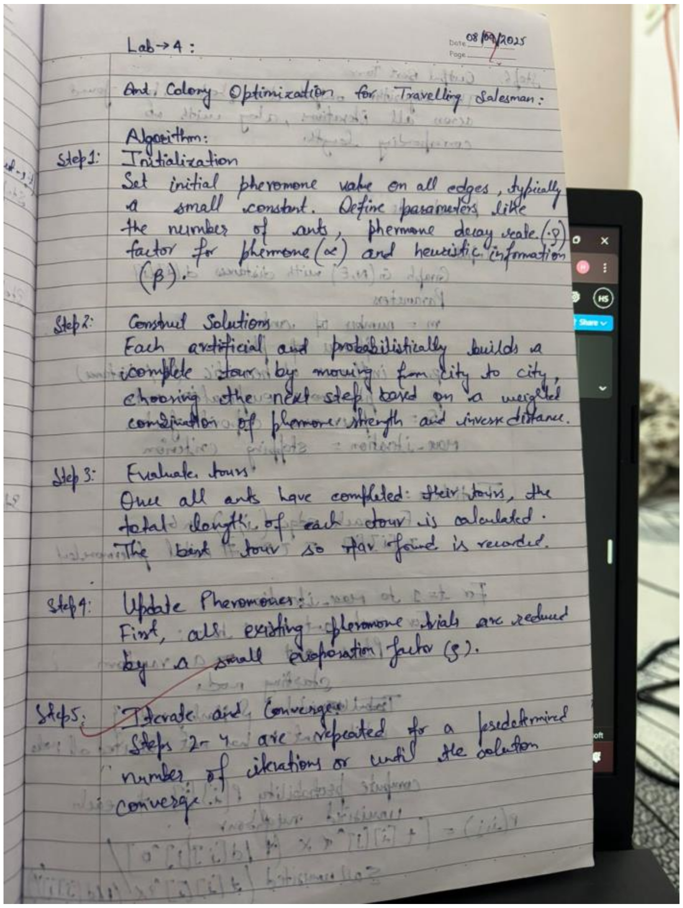
```python
import numpy as np

target_position = np.array([10.0, 10.0])
num_particles = 30
num_iterations = 50
w = 0.7
c1 = 1.5
c2 = 1.5
```

```
v_max = 1.0

positions = np.random.uniform(0, 10, (num_particles, 2))
velocities = np.random.uniform(-1, 1, (num_particles, 2))
personal_best_positions = positions.copy()
personal_best_values = np.full(num_particles, np.inf)
global_best_position = np.zeros(2)
global_best_value = np.inf

def fitness(position, velocity):
    distance = np.linalg.norm(position - target_position)
    smoothness_penalty = np.linalg.norm(velocity)
    return distance + 0.3 * smoothness_penalty

for t in range(num_iterations):
    for i in range(num_particles):
        fit = fitness(positions[i], velocities[i])
        if fit < personal_best_values[i]:
            personal_best_values[i] = fit
            personal_best_positions[i] = positions[i].copy()
        if fit < global_best_value:
            global_best_value = fit
            global_best_position = positions[i].copy()
    for i in range(num_particles):
        r1, r2 = np.random.rand(2)
        velocities[i] = (
            w * velocities[i]
            + c1 * r1 * (personal_best_positions[i] - positions[i])
            + c2 * r2 * (global_best_position - positions[i])
        )
        velocities[i] = np.clip(velocities[i], -v_max, v_max)
        positions[i] += velocities[i]

print("Final optimized position:", global_best_position)
print("Minimum fitness value:", global_best_value)
```

Program 4

Ant Colony Optimization (ACO) for the Travelling SalesMan :

Algorithm:



Lab → 4 :

Date 08/09/2025

Ant Colony Optimization for Travelling Salesman :

Algorithm:

Step 1: Initialization
Set initial pheromone value on all edges, typically a small constant. Define parameters like the number of ants, pheromone decay rate (ρ) factor for pheromone (α) and heuristic information (β).

Step 2: Construct Solutions
Each artificial ant probabilistically builds a complete tour by moving from city to city, choosing the next step based on a weighted combination of pheromone strength and inverse distance.

Step 3: Evaluate tours
Once all ants have completed their tours, the total length of each tour is calculated. The best tour so far found is recorded.

Step 4: Update Pheromones:
First, all existing pheromone trials are reduced by a small evaporation factor (ρ).

Step 5: Iterate and Converge
Steps 2-4 are repeated for a predetermined number of iterations or until the solution converge.

Step 6: Output Best Tour
The algorithm outputs the shortest tour found across all iterations, along with its corresponding length.

Pseudocode:

Input:
Graph $G(N,E)$ with distances $d[i][j]$
Parameters
m = number of ants
α = influence of pheromone
β = influence of heuristic (1 / distance)
ρ = pheromone evaporation rate
Q = phermone deposit factor
max_iteration = stopping criterion

Initialize:
For each edge (i,j) in G
$\tau[i][j] = \tau_0$ # initial pheromone level

For $t = 1$ to max_iterations:
For each ant $k = 1$ to m:
Place ant k on a random starting node
~~Total~~ visited = {starting node}
while ant k has not visited all nodes

compute probability $P[i][j]$ for each unvisited neighbour j:
$$P(i,j) = [\tau[i][j]^\alpha \times (1/d[i][j])^\beta] / \sum_{\text{all unvisited}} (\tau[i][l]^\alpha \times (1/d[i][l])^\beta)$$

choose next node j based on probability $P(i,j)$
Move ant k to node j,
Add j to Tabu_k
Compute tour length L_k for ant k

Update pheromone:
For each edge (i,j):
$\tau[i][j] = (1-\rho) * \tau[i][j]$.

For each ant k:
If ant k used edge (i,j) in its tour
$\tau[i][j] += Q / L\_k$

Keep track of the best solution found so far

Output:
Best tour and its length:

Output:
Distance matrix:

$$\begin{bmatrix} 0 & 10 & 12 & 11 \\ 10 & 0 & 9 & 8 \\ 12 & 9 & 0 & 7 \\ 11 & 8 & 7 & 0 \end{bmatrix}$$

ants $m = 4$
$\alpha = 1.0$
$\beta = 2.0$
$\rho = 0.5$
$Q = 100$
$\tau_0 = 1.0$

heuristic $\eta[i][j] = 1/d[i][j]$

For each edge

Soultion Path:

$A \to B \to D \to C \to A$ $(10 + 8 + 7 + 12)$
Tour length : 37

Code:

```
import numpy as np
import random
import math

# City coordinates
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]
num_cities = len(cities)

# Parameters
num_ants = 20
num_iterations = 100
alpha = 1.0       # pheromone importance
beta = 5.0        # distance importance
rho = 0.5         # pheromone evaporation rate
Q = 100           # pheromone deposit factor

# Distance matrix
dist_matrix = np.zeros((num_cities, num_cities))
for i in range(num_cities):
    for j in range(num_cities):
        if i != j:
            dist_matrix[i][j] = math.sqrt((cities[i][0]-cities[j][0])**2 +
(cities[i][1]-cities[j][1])**2)
        else:
            dist_matrix[i][j] = np.inf

# Initialize pheromone matrix
pheromone = np.ones((num_cities, num_cities))

def probability(i, unvisited, pheromone, dist):
    pher = np.array([pheromone[i][j] ** alpha for j in unvisited])
    vis = np.array([(1.0 / dist[i][j]) ** beta for j in unvisited])
    p = pher * vis
    return p / np.sum(p)

def tour_length(tour):
    return sum(dist_matrix[tour[i]][tour[(i+1)%num_cities]] for i in
range(num_cities))

best_tour = None
best_distance = float("inf")

for _ in range(num_iterations):
    all_tours = []
```

```python
    all_distances = []
    for ant in range(num_ants):
        start = random.randint(0, num_cities - 1)
        tour = [start]
        unvisited = list(range(num_cities))
        unvisited.remove(start)
        while unvisited:
            probs = probability(tour[-1], unvisited, pheromone, dist_matrix)
            next_city = random.choices(unvisited, weights=probs)[0]
            tour.append(next_city)
            unvisited.remove(next_city)
        all_tours.append(tour)
        all_distances.append(tour_length(tour))
        if all_distances[-1] < best_distance:
            best_distance = all_distances[-1]
            best_tour = tour
    pheromone *= (1 - rho)
    for tour, dist in zip(all_tours, all_distances):
        deposit = Q / dist
        for i in range(num_cities):
            a, b = tour[i], tour[(i+1)%num_cities]
            pheromone[a][b] += deposit
            pheromone[b][a] += deposit

print("Best Tour (order of cities):", best_tour)
print("Best Tour Distance:", best_distance)
```

Program 5

Cuckoo Search Algorithms: For JOB SCHEDULING

Algorithm:

LAB-5

Date __/__/__

Page ____

Cuckoo Search Algorithm for Job Scheduling

Algorithm:

1. Initialize population (nests):
   - Generate an initial population of n solutions (nests)
   - Each solution represents a possible job schedule
2. Fitness Evaluation:
   - Calculate the fitness of each schedule

     fitness = 1 / makespan
3. Generate new solutions (Levy Flights).
   - For each cukoo (solution), create a new solution by Levy Flight
4. Evaluates new solution:
   - If the new solution is better than a randomly chosen existing solution, replace it.
5. Abandon fraction of nests (pa):
   - With probability pa, some of the worst solutions are abandoned and replaced with new random solutions.
6. Keep Best Solutions:
7. Repeat until stopping condition.
8. Return the best schedule.

Output:

1 A: {J1, J2} → 5+7 = 12
B: {J3, J4} → 3+9 = 12

Makespan = 12

2 Another Arrangement
A: {J1, J3} → 5+3 = 8
B: {J2, J4} → 7+9 = 16
Makespan = 16

3. A: {J4, J1} → 9+5 = 14
B: {J2, J3} → 7+3 = 10

Makespan = 14

4 A: {J1, J2, J3} → 5+7+3 = 15
B: {J4} → 9
Make span = 15

Best solution:
Machine A: J1 (5) + J2 (7) = 12
Machine B: J3 (3) + J4 (9) = 12

CODE:

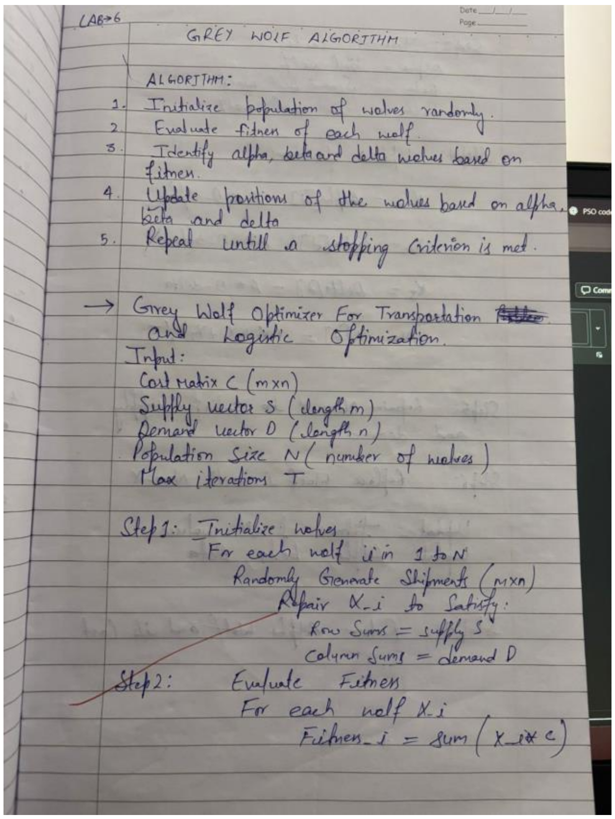
```
import numpy as np
import random

# Number of jobs and machines
num_jobs = 6
num_machines = 3
```

```
# Processing time matrix (rows = jobs, columns = machines)
processing_time = np.array([
    [5, 2, 4],
    [3, 6, 1],
    [4, 3, 5],
    [2, 4, 3],
    [6, 5, 2],
    [7, 3, 4]
])

# Parameters
num_nests = 15
max_iterations = 100
pa = 0.25  # discovery rate of alien eggs

def fitness(schedule):
    machine_loads = [0] * num_machines
    for job, machine in enumerate(schedule):
        machine_loads[machine] += processing_time[job][machine]
    return max(machine_loads)

def levy_flight(Lambda):
    sigma = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
             (np.math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2)))
** (1 / Lambda)
    u = np.random.randn() * sigma
    v = np.random.randn()
    step = u / abs(v) ** (1 / Lambda)
    return step

def get_new_nest(nest):
    new_nest = nest.copy()
    step = int(abs(levy_flight(1.5)) * num_jobs) % num_machines
    job_to_change = random.randint(0, num_jobs - 1)
    new_machine = (new_nest[job_to_change] + step) % num_machines
    new_nest[job_to_change] = new_machine
    return new_nest

# Initialize nests (random assignments of jobs to machines)
nests = [np.random.randint(0, num_machines, num_jobs).tolist() for _ in
range(num_nests)]
fitness_values = [fitness(n) for n in nests]
best_nest = nests[np.argmin(fitness_values)]
best_fitness = min(fitness_values)
```

```python
for t in range(max_iterations):
    new_nests = []
    for nest in nests:
        new_nest = get_new_nest(nest)
        if fitness(new_nest) < fitness(nest):
            new_nests.append(new_nest)
        else:
            new_nests.append(nest)
    nests = new_nests
    # Discovery and randomization
    for i in range(num_nests):
        if random.random() < pa:
            nests[i] = np.random.randint(0, num_machines, num_jobs).tolist()
    fitness_values = [fitness(n) for n in nests]
    current_best = nests[np.argmin(fitness_values)]
    current_fitness = min(fitness_values)
    if current_fitness < best_fitness:
        best_fitness = current_fitness
        best_nest = current_best

print("Best job-to-machine assignment:", best_nest)
print("Minimum makespan (total completion time):", best_fitness)
```

Program 6
Using the Grey Wolf Optimizer (GWO), For Transport and Logistics Transportation

Algorithm:



LAB→6

GREY WOLF ALGORITHM

ALGORITHM:

1. Initialize population of wolves randomly.
2. Evaluate fitnes of each wolf.
3. Identify alpha, beta and delta wolves based on fitnes.
4. Update positions of the wolves based on alpha, beta and delta
5. Repeat untill a stopping Criterion is met.

→ Grey Wolf Optimizer For Transportation ~~[illegible]~~ and Logistic Optimization.

Input:
Cost matrix C (m x n)
Supply vector S (length m)
Demand vector D (length n)
Population Size N (number of wolves)
Max iterations T

Step 1: Initialize wolves
For each wolf i in 1 to N
Randomly Generate Shipments (m x n)
Repair X_i to Satisfy:
Row Sums = supply S
Column Sums = demand D

Step 2: Evaluate Fitness
For each wolf X_i
Fitness_i = sum (X_i * C)

Step 3: Identify Leaders
Alpha = best wolf
Beta = Second best wolf
Delta = third best wolf

Step 4: Update positions

X_i = Alpha[d] − A * D_alpha

X_2 = Beta[d] − A * D_beta

X_3 = Delta[d] − A * D_delta

New_X[d] = $(X_1 + X_2 + X_3)/3$

Step 5: Repair New_X to satisfy supply and demand

Step 6: Replace Wolf i with New_X

Update fitness of all wolves
update Alpha, beta, Delta

Step 7: Return Alpha Wolf and its cost.

Code:

Cost = [[8, 6],
[10, 7]]

Supply = [30, 20]
Demand = [25, 25]

Output:
Best Shipment plan:

$$\begin{bmatrix} 25 & 5 \\ 0 & 20 \end{bmatrix}$$

Best Cost: 370

Code

```python
import numpy as np
import random

# Example: cost matrix (rows = sources, columns = destinations)
cost = np.array([
    [8, 6, 10, 9],
    [9, 12, 13, 7],
    [14, 9, 16, 5]
])

# Supply and demand
supply = [20, 30, 25]
demand = [10, 15, 25, 25]

num_sources = len(supply)
num_destinations = len(demand)
```

```python
num_variables = num_sources * num_destinations

# Grey Wolf Optimizer parameters
num_wolves = 20
max_iter = 100
lb, ub = 0, max(max(supply), max(demand))

# Fitness: total transportation cost + penalty for constraint violation
def fitness(x):
    alloc = x.reshape((num_sources, num_destinations))
    total_cost = np.sum(alloc * cost)
    supply_violation = np.sum(np.abs(np.sum(alloc, axis=1) - supply))
    demand_violation = np.sum(np.abs(np.sum(alloc, axis=0) - demand))
    penalty = 100 * (supply_violation + demand_violation)
    return total_cost + penalty

# Initialize wolves
wolves = np.random.uniform(lb, ub, (num_wolves, num_variables))
fitness_values = np.array([fitness(w) for w in wolves])
alpha, beta, delta = np.argsort(fitness_values)[:3]
alpha_pos, beta_pos, delta_pos = wolves[alpha], wolves[beta], wolves[delta]

for t in range(max_iter):
    a = 2 - t * (2 / max_iter)
    for i in range(num_wolves):
        r1, r2 = np.random.rand(num_variables), np.random.rand(num_variables)
        A1 = 2 * a * r1 - a
        C1 = 2 * r2
        D_alpha = abs(C1 * alpha_pos - wolves[i])
        X1 = alpha_pos - A1 * D_alpha

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

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

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

```
        wolves[i] = np.clip(wolves[i], lb, ub)

    fitness_values = np.array([fitness(w) for w in wolves])
    alpha, beta, delta = np.argsort(fitness_values)[:3]
    alpha_pos, beta_pos, delta_pos = wolves[alpha], wolves[beta], wolves[delta]

best_alloc = alpha_pos.reshape((num_sources, num_destinations))
best_cost = fitness(alpha_pos)

print("Optimal Transportation Plan (allocation matrix):")
print(np.round(best_alloc, 2))
print("Minimum Transportation Cost:", round(best_cost, 2))
```

Program 7
Parallel Cellular Algorithm for Epidemic Spread

Algorithm:

13/10/25

Date: 9

LAB→7 Parallel Cellular Algorithm:

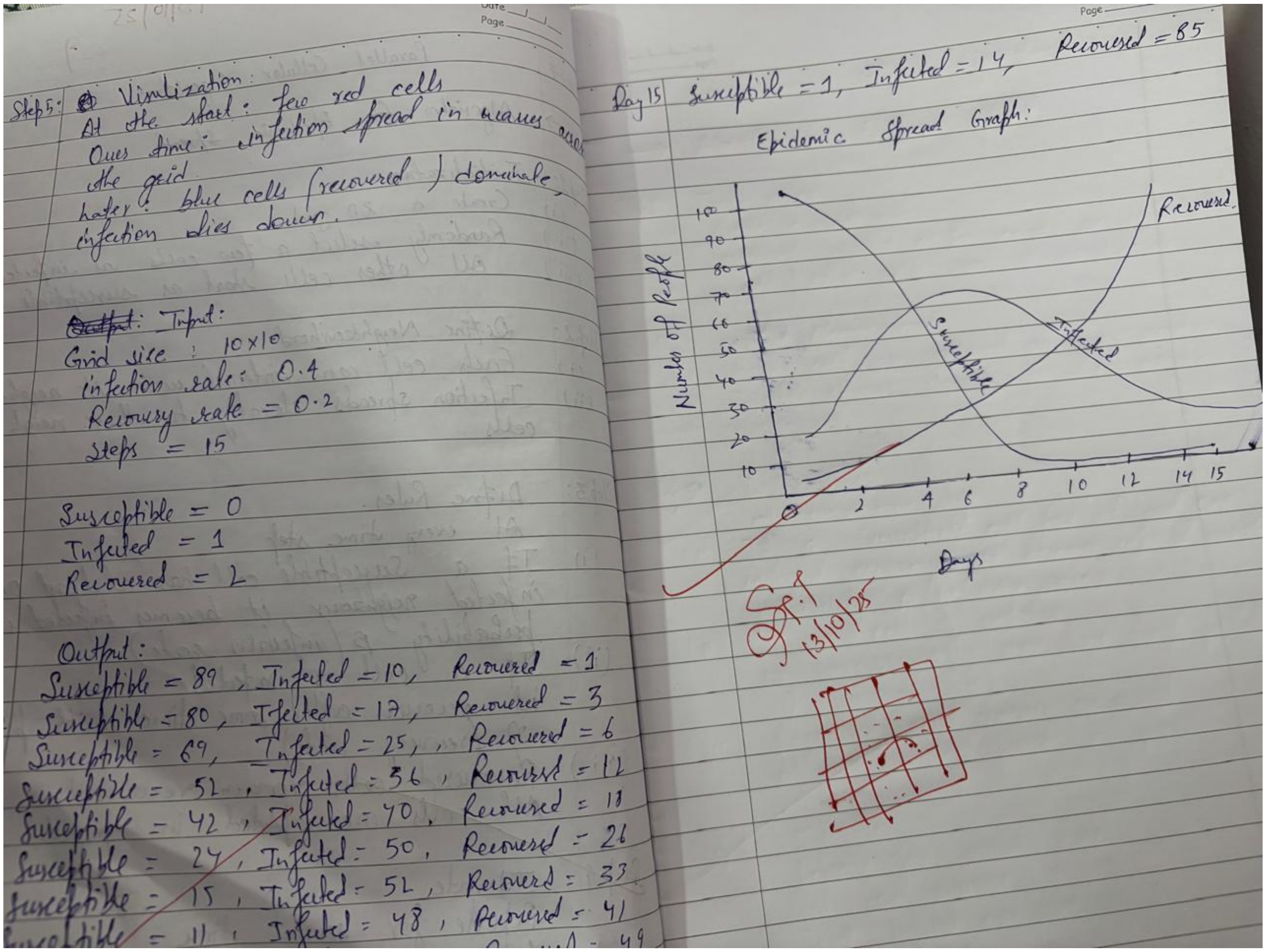
→ Algorithm for Epidemic Spread

Step 1: Initialization
(i) Create a 2D grid.
(ii) Randomly select a few cells as infected
(iii) All other cells start as susceptible.

Step 2: Define Neighbourhood
(i) Each cell can interact with its neighbours
(ii) Infection spreads locally to these neighbouring cells.

Step 3: Define Rules
At every time step
(i) If a susceptible cell has at least one infected neighbour it becomes infected with probability β (infection rate)
(ii) If a cell is infected it recovers after some time with probability γ (recovery rate).
(iii) Recovered cells remain recovered
All cells are updated in parallel

Step 4: Iterate
Repeate for several time.
You can measure.
Total infected over time
infection peaks
Speed and pattern spread


Step 5: Visualization:
At the start: few red cells
Over time: infection spread in waves across the grid
Later: blue cells (recovered) dominate, infection dies down.

~~Output~~: Input:
Grid size : 10×10
infection rate: 0.4
Recovery rate = 0.2
Steps = 15

Susceptible = 0
Infected = 1
Recovered = 2

Output:
Susceptible = 89, Infected = 10, Recovered = 1
Susceptible = 80, Infected = 17, Recovered = 3
Susceptible = 69, Infected = 25, Recovered = 6
Susceptible = 52, Infected = 36, Recovered = 12
Susceptible = 42, Infected = 40, Recovered = 18
Susceptible = 24, Infected = 50, Recovered = 26
Susceptible = 15, Infected = 52, Recovered = 33
Susceptible = 11, Infected = 48, Recovered = 41

Day 15 Susceptible = 1, Infected = 14, Recovered = 85

Epidemic Spread Graph:

Code:

```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Example epidemic data (rows = days, columns = S, I, R)
data = pd.DataFrame({
    'Susceptible': [89, 80, 69, 52, 35, 20, 12],
    'Infected': [10, 17, 25, 36, 42, 35, 22],
    'Recovered': [1, 3, 6, 12, 23, 45, 66]
})

# Standardize the data
data_std = (data - data.mean()) / data.std()
```

```python
# Apply PCA
pca = PCA(n_components=2)
principal_components = pca.fit_transform(data_std)
pca_df = pd.DataFrame(principal_components, columns=['PC1', 'PC2'])

# Explained variance ratio
print("Explained variance ratio:", pca.explained_variance_ratio_)

# Combine results
final_df = pd.concat([data, pca_df], axis=1)
print(final_df)

# Plot (optional)
plt.scatter(pca_df['PC1'], pca_df['PC2'], color='blue')
plt.title('PCA of Epidemic Spread')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True)
plt.show()
```