VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Utkrisht Umang (1BM23CS355)

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

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Utkrisht Umang (1BM23CS355),** 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/utk1college/BIS

Program 1

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

Algorithm:

PAGE EDGE

DATE: 18/01/2025

Genetic Algorithm

1.) Selecting Initial population
2.) Calculate the fitness
3.) Selecting the mating Pool
4.) Crossover
5.) Mutation

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

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

1. Ex: $x \to 0-31$

2.

String No.	Initial Population	x value	Fitness $f(x) = x^2$	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	181	0.3126	31.26	1.25	1
Sum			1155	1.0	100	4	
Average			288.76	0.25	25	1	
Maximum			625	0.5411	54.11	2.16	

5. Selecting Mating Pool:

String No	Mating Pool	Crossover Point	Offspring after Crossover	X-value	Fitness $f(x)=x^2$
1	0110$\boxed{0}$	4	01101	13	169
2	1100$\boxed{1}$		11000	24	576
3	11$\boxed{001}$	2	11011	27	729
4	10$\boxed{011}$		10001	17	289
Sum					1763
Max.					440.75
Avg.					729

Crossover

Crossover point is chosen randomly.

Mutation: chosen randomly

String No	Offspring after Crossover	Mutation Chromosome	Offspring after Chromosome	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
Avg.					636.5
Max.					841

→ LAB-1

Job Scheduling with Genetic Algorithms:-

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

Algorithm / Pseudocode

1. Define the problem.
   - Jobs with processing times
   - Machines with capacity (optional).
2. Initialize parameters → Set P, Pc, Pm and G
3. Initial population → Generate random job sequences.
4. Fitness function → Fitness = 1/makespan of schedule
5. Selection → Choose parents using roulette/tournament
6. Crossover → Apply Order Crossover (OX) on parents
7. Mutation → Swap two jobs randomly in a chromosome.
8. Iteration → Repeat evaluate - select - crossover - mutate for G generations.
9. Output → Best chromosome gives near optimal schedule.

Output:-

Best Job Order: [2, 1, 4, 5, 3, 0]

Job Times: [7, 2, 9, 4, 5, 3]

Total Completion Time (Makespan): 30

Code:

```
import random

jobs = [3, 2, 7, 5, 9, 4]  # processing times of jobs
num_jobs = len(jobs)
population_size = 20
generations = 100
crossover_rate = 0.8
mutation_rate = 0.2

# ------------------------------
# Fitness Function (Makespan)
# ------------------------------
def fitness(chromosome):
    time = 0
    for job in chromosome:
        time += jobs[job]
    return 1 / time   # smaller time → higher fitness

def initial_population():
    population = []
    for _ in range(population_size):
        chromosome = list(range(num_jobs))
        random.shuffle(chromosome)
        population.append(chromosome)
    return population
def selection(population):
    contenders = random.sample(population, 3)
    contenders.sort(key=lambda chromo: fitness(chromo), reverse=True)
    return contenders[0]
def crossover(p1, p2):
    if random.random() < crossover_rate:
        a, b = sorted(random.sample(range(num_jobs), 2))
        child = [-1] * num_jobs
        child[a:b] = p1[a:b]
        fill = [x for x in p2 if x not in child]
        j = 0
        for i in range(num_jobs):
            if child[i] == -1:
                child[i] = fill[j]
                j += 1
        return child
    return p1[:]  # no crossover → copy parent
def mutate(chromosome):
    if random.random() < mutation_rate:
        a, b = random.sample(range(num_jobs), 2)
        chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
```

```
    return chromosome
population = initial_population()
best_solution = None
best_fit = -1

for gen 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

    # Track best
    for chromo in population:
        fit = fitness(chromo)
        if fit > best_fit:
            best_fit = fit
            best_solution = chromo
print("Best Job Order:", best_solution)
print("Job Times:", [jobs[j] for j in best_solution])
print("Total Completion Time (Makespan):", sum(jobs[j] for j in best_solution))
```

Output:

```
Best Job Order: [3, 4, 0, 2, 1, 5]
Job Times: [5, 9, 3, 7, 2, 4]
Total Completion Time (Makespan): 30
```

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:

25/08/2025

→ Lab 2

Gene Expression Algorithms:-

It can solve Travelling Salesman Problem by encoding city sequences as chromosomes and evolving them through selection, crossover, mutation, and expression to find the shortest possible tour.

1. Problem → N cities given, find shortest possible tour visiting each once.
2. Parameters → Set P (population), Pc (crossover), Pm (mutation), G (generations).
3. Population → Start with random permutations of city order.
4. Fitness → fitness = 1/tour distance (shorter tour = better).
5. Selection → Choose fitter tours using tournament or roulette selection.
6. Crossover → Use Order Crossover (OX) to mix parent tours.
7. Mutation → Swap or reverse cities to keep diversity.
8. Gene Expression → Convert chromosome into a valid city tour.
9. Iterate → Repeat evaluation → selection → crossover → mutation → expression for G generations.
10. Output → Best chromosome = near-optimal TSP route.

PAGE EDGE
DATE: / /

// Output:-

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

Best Tour(order of cities): [3,1,0,2,4]
Best Tour Distance = 22.35103

8/10
25/8

Code:

```
import random
import math

# -----------------------------
# Problem: TSP cities
# -----------------------------
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]  # coordinates
num_cities = len(cities)

# Parameters
population_size = 30
generations = 200
crossover_rate = 0.8
mutation_rate = 0.2

# -----------------------------
# Distance Function
# -----------------------------
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

# ----------------------------
# Fitness Function
# ----------------------------
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")

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

    # Track best solution
    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)
```

Output:

```
Best Tour (order of cities): [4, 2, 0, 1, 3]
Best Tour Distance: 22.35103276995244
```

Program 3

Portfolio Optimization (Selecting assets) using Particle Swarm Optimization is about choosing how much money to allocate to different assets (stocks, bonds, etc.) to maximize expected return while minimizing risk (variance).

Algorithm:

04/09/2025

→ LAB-3

Particle Swarm Optimization for Function Optimization:

→ Portfolio Optimization (selecting assets) is about choosing how much money to allocate to different assets (stocks, bonds, etc.) to maximize expected return while minimizing risk (variance).

1. Initialize particles with random portfolio weights (positions) and velocities.
2. Evaluate fitness (Sharpe Ratio) of each particle.
3. Set each particle's personal best (P_{best}) and the swarm's global best (g_{best}).
4. Repeat until max iterations:
   - Update velocity:
     $v = w * v + c_1 * r_1 (P_{best} - pos) + c_2 * r_2 (g_{best} - pos)$
   - Update position (weights):
     pos = pos + v
     normalize (pos) # weights ≥ 0, sum=1
   - Re-evaluate fitness and update P_{best}, g_{best} if improved.
5. Return g_{best} position as the optimal portfolio.

11 Output:-

Expected returns = [0.12, 0.18, 0.15, 0.10].
The code runs 100 iterations by default.
Optimal Portfolio Weights: [0.44097, 0.20835, 0.28230, 0.06827].
Best Sharpe Ratio : 1.77560983

Code:

```
import numpy as np

# ---------- Step 1: Define Problem (Portfolio Optimization) ----------
# Expected returns for 4 assets (example data)
returns = np.array([0.12, 0.18, 0.15, 0.10])

# Covariance matrix of returns (risk measure)
cov_matrix = np.array([
    [0.010, 0.002, 0.001, 0.003],
    [0.002, 0.030, 0.002, 0.004],
    [0.001, 0.002, 0.020, 0.002],
    [0.003, 0.004, 0.002, 0.025]
])

# Fitness function: Sharpe ratio (maximize return / risk)
def fitness(weights):
    weights = np.array(weights)
    portfolio_return = np.dot(weights, returns)
    portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
    if portfolio_risk == 0:  # avoid division by zero
        return -999
    return portfolio_return / portfolio_risk

# ---------- Step 2: Initialize PSO Parameters ----------
num_particles = 30
num_assets = len(returns)
iterations = 100

w = 0.7      # inertia weight
c1 = 1.5     # cognitive coefficient
c2 = 1.5     # social coefficient

# ---------- Step 3: Initialize Particles ----------
positions = np.random.dirichlet(np.ones(num_assets), size=num_particles)  # weights sum=1
velocities = np.random.rand(num_particles, num_assets) * 0.1

personal_best_positions = positions.copy()
personal_best_scores = np.array([fitness(p) for p in positions])

global_best_position = personal_best_positions[np.argmax(personal_best_scores)]
global_best_score = np.max(personal_best_scores)

# ---------- Step 4: Main Loop ----------
for _ in range(iterations):
    for i in range(num_particles):
        # Update velocity
        r1, r2 = np.random.rand(num_assets), np.random.rand(num_assets)
```

```
        velocities[i] = (w * velocities[i]
                         + c1 * r1 * (personal_best_positions[i] - positions[i])
                         + c2 * r2 * (global_best_position - positions[i]))

        # Update position (weights must be valid portfolio)
        positions[i] += velocities[i]
        positions[i] = np.maximum(positions[i], 0)     # no negative weights
        positions[i] /= np.sum(positions[i])           # normalize to sum=1

        # Evaluate fitness
        score = fitness(positions[i])

        # Update personal best
        if score > personal_best_scores[i]:
            personal_best_scores[i] = score
            personal_best_positions[i] = positions[i].copy()

        # Update global best
        if score > global_best_score:
            global_best_score = score
            global_best_position = positions[i].copy()

# ---------- Step 5: Output Result ----------
print("Optimal Portfolio Weights:", global_best_position)
print("Best Sharpe Ratio:", global_best_score)
```

Output:

```
Optimal Portfolio Weights: [0.44097408 0.20835576 0.2823928  0.06827736]
Best Sharpe Ratio: 1.7756098324447378
```

Program 4

Ant Colony Optimization (ACO) for the Vehicle Routing Problem (VRP): It involves finding optimal routes for multiple vehicles to deliver goods to a set of customers from a central depot.

Algorithm:

08/09/2025

→ LAB-4

Ant Colony Optimization (ACO) for the Vehicle Routing Problem (VRP):

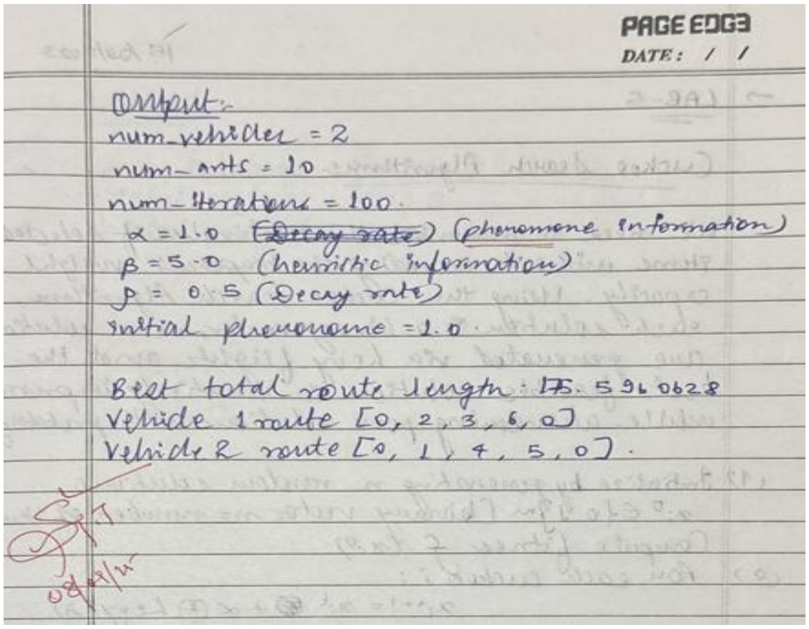
→ It involves finding optimal routes for multiple vehicles to deliver goods to a set of customers from a central depot.

1. Initialize pheromone τ_{ij} and heuristic $\eta_{ij} = 1/d_{ij}$.
2. Set parameters: ants m, pheromone weight α, heuristic weight β, evaporation ρ, iterations. $\alpha \to$ influence of pheromone; $\beta \to$ influence of heuristic information
3. For each iteration:
   - Each ant builds routes from depot by choosing next city j with probability:

     $$p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{k \in \text{unvisited}} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}}$$

     until all customers visited; return to depot.
4. Calculate total route length L for each ant.
5. Evaporate pheromone, $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$
6. Deposit pheromone on best routes:
7. Repeat $\tau_{ij} \leftarrow \tau_{ij} + \frac{1}{L}$

   until stopping condition.
8. Output best routes and length.


Output:-
num_vehicles = 2
num_ants = 10
num_iterations = 100.
α = 1.0 ~~(Decay rate)~~ (pheromone information)
β = 5.0 (heuristic information)
ρ = 0.5 (Decay rate)
initial pheromone = 1.0.

Best total route length: 175.5960628
Vehicle 1 route [0, 2, 3, 6, 0]
Vehicle 2 route [0, 1, 4, 5, 0].

Code:

```
import numpy as np
import random

# Coordinates of depot + customers (0 is depot)
coords = np.array([
    [40, 50],  # depot
    [45, 68], [50, 30], [55, 20], [60, 80], [65, 60], [70, 40]
])

num_vehicles = 2
num_ants = 10
num_iterations = 100
alpha = 1.0  # pheromone importance
beta = 5.0   # heuristic importance (inverse distance)
rho = 0.5    # pheromone evaporation rate
initial_pheromone = 1.0

num_cities = len(coords)

# Distance matrix
dist_matrix = np.sqrt(((coords[:, None] - coords[None, :])**2).sum(axis=2))

# Heuristic matrix (inverse distance), avoid division by zero
heuristic = 1 / (dist_matrix + np.diag([np.inf]*num_cities))

# Initialize pheromone trails
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
```

```
def choose_next_city(current_city, unvisited, pheromone, heuristic):
    pheromone_vals = pheromone[current_city][unvisited] ** alpha
    heuristic_vals = heuristic[current_city][unvisited] ** beta
    probs = pheromone_vals * heuristic_vals
    probs /= probs.sum()
    return np.random.choice(unvisited, p=probs)

def construct_solution():
    routes = [[] for _ in range(num_vehicles)]
    unvisited = set(range(1, num_cities))  # customers only
    for v in range(num_vehicles):
        routes[v].append(0)  # start from depot

    while unvisited:
        for v in range(num_vehicles):
            current_city = routes[v][-1]
            candidates = list(unvisited)
            if not candidates:
                break
            next_city = choose_next_city(current_city, candidates, pheromone, heuristic)
            routes[v].append(next_city)
            unvisited.remove(next_city)
            if not unvisited:
                break

    # Return to depot
    for v in range(num_vehicles):
        routes[v].append(0)
    return routes

def route_length(route):
    length = 0
    for i in range(len(route)-1):
        length += dist_matrix[route[i], route[i+1]]
    return length

best_routes = None
best_length = float('inf')

for iteration in range(num_iterations):
    all_routes = []
    all_lengths = []

    for _ in range(num_ants):
        routes = construct_solution()
        total_length = sum(route_length(r) for r in routes)
        all_routes.append(routes)
        all_lengths.append(total_length)
```

```
        if total_length < best_length:
            best_length = total_length
            best_routes = routes

    # Pheromone evaporation
    pheromone *= (1 - rho)

    # Pheromone update (only best ant deposits pheromone)
    for route in best_routes:
        for i in range(len(route)-1):
            from_city = route[i]
            to_city = route[i+1]
            pheromone[from_city][to_city] += 1 / best_length
            pheromone[to_city][from_city] += 1 / best_length

print("Best total route length:", best_length)
for v, route in enumerate(best_routes):
    print(f"Vehicle {v+1} route: {route}")
```

Output:

```
Best total route length: 175.5960628325094
Vehicle 1 route: [0, np.int64(1), np.int64(4), np.int64(5), 0]
Vehicle 2 route: [0, np.int64(2), np.int64(3), np.int64(6), 0]
```

Program 5

Cuckoo Search Algorithms: We need to maximize the total value of selected items without exceeding the knapsack's weight capacity. Using the Cuckoo Search Algorithm, each solution is a binary vector, new solutions are generated via Lévy flights, and the best feasible solution is iteratively improved while abandoning poor solutions with a probability.

Algorithm:

15/09/2025

→ LAB-5

Cuckoo Search Algorithms:-

We need to maximize the total value of selected items without exceeding the knapsack's weight capacity. Using the Cuckoo Search Algorithm, each solution is a binary vector, new solutions are generated via Levy flights, and the best feasible solution is iteratively improved while abandoning poor solutions with probability.

(1) Initialize by generating n random solutions $x_i^0 \in \{0,1\}^m$ (binary vector, n = number of items). Compute fitness $f(x_i^0)$.

(2) For each cuckoo i:

$$x_i^{t+1} = x_i^t + \alpha \oplus Levy(\lambda)$$

(3) Compute fitness:-

$$f(x_i^{t+1}) = \Sigma v_j x_{ij}, \text{ if } \Sigma w_j x_{ij} \le W$$

- If weight exceeds W, assign very low fitness

(4) If $f(x_i^{t+1}) > f(x_j^t)$ for a random nest j:

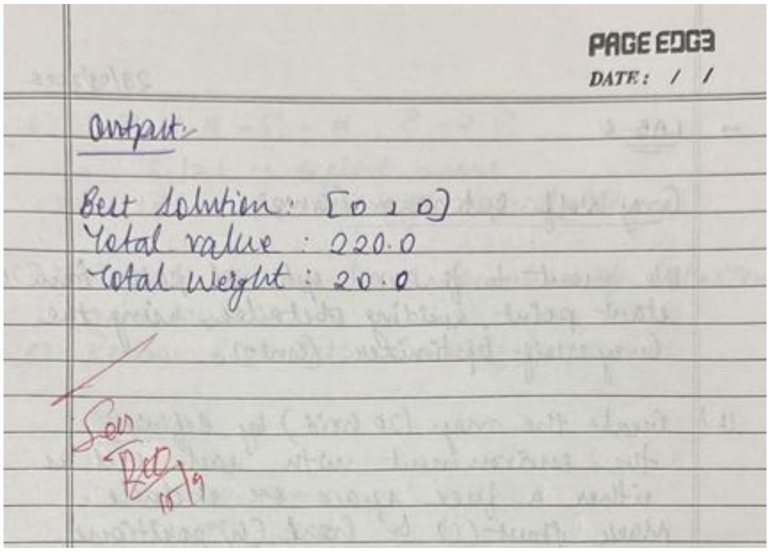
$$x_j^t \leftarrow x_i^{t+1}$$

(5) With probability P_a, abandon a fraction of worst nests and replace them with new random solutions

(6) Track best so far :- $x^* = \arg\max(f(x_i^t))$

(7) Repeat steps 2-6 until max-iterations.

(8) Output best solution x^* (items chosen, max-value within capacity).



Code:

```
import numpy as np
import random

# ---------------- Knapsack Problem Setup ----------------
# Example items: (value, weight)
items = [(60, 10), (100, 20), (120, 30)]
capacity = 50
n = len(items)

def fitness(solution):
    total_value = total_weight = 0
    for i in range(n):
        if solution[i] == 1:
            total_value += items[i][0]
            total_weight += items[i][1]
    if total_weight > capacity:
        return 0  # invalid solution
    return total_value

# ---------------- Cuckoo Search Algorithm ----------------
def levy_flight(Lambda):
    u = np.random.normal(0, 1) * np.power(abs(np.random.normal(0, 1)), -1.0 / Lambda)
    v = np.random.normal(0, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step

def get_random_solution():
    return [random.randint(0, 1) for _ in range(n)]

def cuckoo_search(num_nests=10, pa=0.25, max_iter=100):
```

```
    nests = [get_random_solution() for _ in range(num_nests)]
    best = max(nests, key=fitness)

    for _ in range(max_iter):
        # Generate new solution via Levy flight
        cuckoo = best[:]
        step = int(abs(round(levy_flight(1.5)))) % n
        pos = random.randint(0, n-1)
        cuckoo[pos] = 1 - cuckoo[pos]  # flip bit

        # Replace a random nest if better
        j = random.randint(0, num_nests-1)
        if fitness(cuckoo) > fitness(nests[j]):
            nests[j] = cuckoo

        # Abandon some nests with probability pa
        for i in range(num_nests):
            if random.random() < pa:
                nests[i] = get_random_solution()

        # Update best
        best = max(nests, key=fitness)

    return best, fitness(best)

# ---------------- Run the algorithm ----------------
solution, value = cuckoo_search()
print("Best solution:", solution)
print("Total value:", value)
```

Output:

```
Best solution: [0, 1, 1]
Total value: 220
```

Program 6

Using the Grey Wolf Optimizer (GWO), we aim to find the shortest, obstacle-free path by modeling the search agents (wolves) to iteratively converge toward the best position (path node) in the environment. The algorithm simulates the grey wolves' hunting hierarchy and encircling behavior to efficiently navigate the space from the start point.

Algorithm:

29/09/2025

→ LAB-6

Grey Wolf Optimizer (GWO)

We want to find an optimal path from a start point, avoiding obstacles, using the Grey Wolf Optimizer (GWO).

(1) Create the map (2D Grid) by defining the environment with each cell as either a free space or obstacle. Mark Start (S) & Goal (G) positions.

(2) Initialize population of paths (Wolves).

(3) Define a fitness function $f(x_i)$ for path (x_i): $f(x_i) = \text{length}(x_i) + P_{obs} \cdot n_{obs} + P_{miss} \cdot I_{\text{goal not reached}}$

(4) Find α, β & δ Wolves & Assign:

X_α : best path (lowest fitness)

X_β : 2nd best path

X_δ : 3rd best path

(5) Update each path:

$$\vec{D}_\alpha = |\vec{C}_1 \cdot \vec{X}_\alpha - \vec{X}_i| \quad \text{(same for } \beta, \delta)$$

Calculate the new position:

$$\vec{X}_1 = \vec{X}_\alpha - \vec{A}_1 \cdot \vec{D}_\alpha \quad \text{(same for } X_2 \,\&\, X_3)$$

Average to get the new positions:-

$$\vec{X}_i(t+1) = \frac{1}{3}(\vec{X}_1 + \vec{X}_2 + \vec{X}_3).$$

PAGE EDGE

DATE: / /

(6) $\vec{A} = 2 \cdot \vec{a} \cdot \vec{r}_1 - \vec{a}$, $\vec{C} = 2 \cdot \vec{r}_2$

$|\vec{A}| > 1 \rightarrow$ explore more

$|\vec{A}| < 1 \rightarrow$ converge to solution.

(7) Repeat for a fixed number of solutions iterations.

(8) Return Best Path (α Wolf) i.e. X_α

Output:-

```
S * * * *
. . # . *
. . # . *
. . # . *
. . . . G
```

Best Path (coordinates):-

[(0,0), (1,0), (2,0), (3,0), (4,0), (4,1), (4,2), (4,3), (4,4)]

Path length : 9

Cost : 9

Code:

```
import numpy as np
import random

# === Grid setup ===
GRID_SIZE = 5
START = (0, 0)
GOAL = (4, 4)
OBSTACLES = [(2, i) for i in range(1, 4)]  # Vertical wall in column 2, rows 1 to 3

# === Parameters ===
POP_SIZE = 10
MAX_ITER = 50
PATH_LENGTH = 20  # fewer steps needed for small grid

# === Helper Functions ===

def is_valid(pos):
    x, y = pos
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and pos not in OBSTACLES

def move_toward_goal(current):
    moves = [(0,1), (1,0), (0,-1), (-1,0)]
    random.shuffle(moves)
```

```python
    cx, cy = current
    gx, gy = GOAL
    moves.sort(key=lambda m: abs((cx + m[0]) - gx) + abs((cy + m[1]) - gy))
    for dx, dy in moves:
        new_pos = (cx + dx, cy + dy)
        if is_valid(new_pos):
            return new_pos
    return current

def generate_random_path():
    path = [START]
    visited = set(path)
    current = START
    for _ in range(PATH_LENGTH):
        current = move_toward_goal(current)
        if current in visited:
            continue
        path.append(current)
        visited.add(current)
        if current == GOAL:
            break
    return path

def path_cost(path):
    cost = len(path)
    if path[-1] != GOAL:
        dist = abs(path[-1][0] - GOAL[0]) + abs(path[-1][1] - GOAL[1])
        cost += 100 + dist
    for pos in path:
        if pos in OBSTACLES:
            cost += 50
    return cost

# === GWO Optimization ===

def gwo_optimize():
    wolves = [generate_random_path() for _ in range(POP_SIZE)]

    for iteration in range(MAX_ITER):
        wolves.sort(key=path_cost)
        alpha, beta, delta = wolves[0], wolves[1], wolves[2]
        a = 2 - iteration * (2 / MAX_ITER)

        for i in range(3, POP_SIZE):
            new_path = []
            for j in range(min(len(alpha), len(wolves[i]), PATH_LENGTH)):
                A = 2 * a * random.random() - a
                C = 2 * random.random()
                x_alpha = np.array(alpha[j])
```

```python
                x_wolf = np.array(wolves[i][j])
                D_alpha = abs(C * x_alpha - x_wolf)
                X1 = x_alpha - A * D_alpha

                A = 2 * a * random.random() - a
                C = 2 * random.random()
                x_beta = np.array(beta[j])
                D_beta = abs(C * x_beta - x_wolf)
                X2 = x_beta - A * D_beta

                A = 2 * a * random.random() - a
                C = 2 * random.random()
                x_delta = np.array(delta[j])
                D_delta = abs(C * x_delta - x_wolf)
                X3 = x_delta - A * D_delta

                X_new = (X1 + X2 + X3) / 3
                X_new = tuple(map(int, np.clip(np.round(X_new), 0, GRID_SIZE - 1)))

                if is_valid(X_new):
                    new_path.append(X_new)
                else:
                    if new_path:
                        new_path.append(move_toward_goal(new_path[-1]))
                    else:
                        new_path.append(move_toward_goal(START))
            wolves[i] = new_path

    best_path = sorted(wolves, key=path_cost)[0]
    return best_path

# === Textual Output ===

def print_grid(path):
    grid = [["." for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

    for x, y in OBSTACLES:
        grid[y][x] = "#"  # Obstacle

    for x, y in path:
        if (x, y) != START and (x, y) != GOAL and grid[y][x] != "#":
            grid[y][x] = "*"

    sx, sy = START
    gx, gy = GOAL
    grid[sy][sx] = "S"
    grid[gy][gx] = "G"

    print("\n=== GWO Path Grid ===")
```

```
    for row in grid:
        print(" ".join(row))

    print("\nBest Path (coordinates):")
    print(path)

    print(f"\nPath Length: {len(path)}")
    print(f"Cost: {path_cost(path)}")

# === Run ===

best = gwo_optimize()
print_grid(best)
```

Output:

```
=== GWO Path Grid ===
S . . . .
* * # . .
. * # . .
. * # . .
. * * * G

Best Path (coordinates):
[(0, 0), (0, 1), (1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4)]

Path Length: 9
Cost: 9
```

Program 7

The task is to perform edge detection or noise reduction in an image using Parallel Cellular Automata (PCA), where each pixel (cell) interacts with its neighbors to enhance edges or reduce noise iteratively.

Algorithm:

13/10/2025

→ LAB-7

PCA

The task is to perform edge detection or noise reduction in an image using Parallel Cellular Automata (PCA), where each pixel (cell) interacts with its neighbors to enhance edges or reduce noise iteratively.

```
function cellular_automata (image, iterations,
        threshold):
    grid = image.copy()   # Initialize Grid
    neighbours = [(-1,1), (-1,0), (-1,1), (0,-1), (0,0),
                  (0,1), (1,-1), (1,0), (1,1)]
    for iteration in range (iterations):
        updated_grid = grid.copy()
        for i in range (1, M-1):
            for j in range (1, N-1):
                pixel = grid [i, j]
                neighbor_vals = [grid [i+di, j+dj]
                                 for (di, dj) in neighbours]

                # Edge detection
                if (max (neighbor_vals) -
                        min (neighbor_vals)) > threshold:
                    updated_grid [i, j] = 255
                else:
                    updated_grid [i, j] = sum (neighbor_vals)
                                          / 8

        grid = updated_grid
    return grid
```

PAGE EDGE
DATE: / /

Input:-

- image : MxN matrix of pixel values (grayscal
- iterations : No of iterations to apply the algorithm
- threshold : For edge detection, defines how large a difference must be to classify an edge.

Output:-

Original Image (Pixel values):

[69 29 109 179 17]
[27 24 168 205 178]
[58 136 53 201 152]
[153 151 35 71 185]
[230 169 136 210 5]

Processed Image (Pixel values):

[69 29 109 179 17]
[27 255 255 255 178]
[58 255 30 255 152]
[153 255 255 255 185]
[230 169 136 210 5]

Code:

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

# Function for Cellular Automata (Edge Detection or Noise Reduction)
def cellular_automata(image, iterations=10, threshold=30):
    grid = image.copy()  # Initialize grid (image as 2D array)
    neighbors = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)]

    for iteration in range(iterations):
        updated_grid = grid.copy()

        for i in range(1, len(grid) - 1):  # Loop through pixels (excluding borders)
```

```
            for j in range(1, len(grid[0]) - 1):
                pixel = grid[i, j]
                neighbor_vals = [grid[i+di, j+dj] for (di, dj) in neighbors]

                # Edge detection: large difference with neighbors indicates edge
                if max(neighbor_vals) - min(neighbor_vals) > threshold:
                    updated_grid[i, j] = 255  # Edge pixel
                else:
                    # Noise reduction: average with neighbors for smoothing
                    new_pixel_value = sum(np.clip(neighbor_vals, 0, 255)) // 8  # Clipping before averaging

                    # Clip the new pixel value to the range 0-255
                    updated_grid[i, j] = np.clip(new_pixel_value, 0, 255)

        grid = updated_grid  # Update the grid with new values

    return grid  # Output updated image

# Set numpy to ignore overflow warnings
np.seterr(over='ignore')

# Generate a smaller dummy grayscale image (random noise)
# Create a 5x5 pixel image with random values between 0 and 255
image = np.random.randint(0, 256, (5, 5), dtype=np.uint8)

# Print the original image
print("Original Image (Pixel Values):")
for row in image:
    print(row)

# Apply the cellular automata algorithm
iterations = 10
threshold = 30
processed_image = cellular_automata(image, iterations, threshold)

# Print the processed image
print("\nProcessed Image (Pixel Values):")
for row in processed_image:
    print(row)

# Visualize the images using matplotlib
plt.figure(figsize=(8,4))

plt.subplot(1,2,1)
plt.title('Original Image')
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(1,2,2)
```

```
plt.title('Processed Image')
plt.imshow(processed_image, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.tight_layout()
plt.show()
```

Output:

```
Original Image (Pixel Values):
[229  27  88 154 254]
[125  20  90 185 144]
[200 161  35  19  61]
[108 251   0 187  28]
[ 12  50  54 119 225]

Processed Image (Pixel Values):
[229  27  88 154 254]
[125 255 255 255 144]
[200 255  30 255  61]
[108 255 255 255  28]
[ 12  50  54 119 225]
```