VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Bhoomi Udedh(1BM23CS066)

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 **Bhoomi Udedh(1BM23CS066),** who is a 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.

Mayanka Gupta Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.

Github Link:

https://github.com/bhoomiudedh/BIS/tree/main

Program 1 : Genetic Algorithm

Problem statement:

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems.

Algorithm:

Genetic Algorithm

1) Selecting Initial Population
2) Calculate the fitness
3) Selecting the mating pool
4) Crossover
5) Mutation

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

$$\text{Expected output} = \frac{f(x)}{\text{Avg.}(\Sigma f(x))} = \frac{144}{283.75}$$

Ex. (1) $x \to 0 - 31$

(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	$\boxed{625}$	0.5411	54.11	2.16	2
3	00101	5	25	0.0216	2.16	0.08	0
4	10011	19	(361) 181	0.3126	31.26	1.25	1
Sum			(1355)	1.0	100	4	
Average			283.75	0.25	25	1	
Mutation			625	0.5411	54.11	2.16	

(3) Selecting Mating Pool

String No.	Mating Pool	Crossover 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
					1763
					440.75
					729

Crossover

Crossover point is chosen randomly

Mutation

= 164
1155
= 0.1247
0.4987

Actual count
1
2
0

Mutation.

String No.	Offspring after crossover	Mutation chromosome	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
Max mutation					841

Sample Output for Genetic Algorithm for Optimization Problems Using Python:-

Gen 0 : Best $x = 7.6382$, $f(x) = 6.9351$
Gen 1 : Best $x = 8.2073$, $f(x) = 7.5629$
:
Gen 49 : Best $x = 9.1075$, $f(x) = 8.0782$
Best solution found:
$x = 9.1075$
$f(x) = 8.0782$

Overview of Problem:-

fitness → $f(x) = \sin(x) \cdot x$
Domain → $[0, 10]$
Generations = 50.

Code:

```
import random def
fitness(x):
return x**2    def
int_to_bin(x):
   return format(x, '05b') def bin_to_int(b):     return
int(b, 2) def tournament_selection(pop, k=3):
selected = random.sample(pop, k)
selected.sort(key=lambda x: fitness(x), reverse=True)
return selected[0] def crossover(p1, p2):
   b1, b2 = int_to_bin(p1), int_to_bin(p2)
point = random.randint(1, 4)     child1 =
bin_to_int(b1[:point] + b2[point:])     child2 =
bin_to_int(b2[:point] + b1[point:])     return
child1, child2 def mutate(x,
mutation_rate=0.1):     if random.random() <
mutation_rate:
      b = list(int_to_bin(x))        pos =
random.randint(0, 4)        b[pos] = '1'
if b[pos] == '0' else '0'        return
bin_to_int("".join(b))     return x
def   genetic_algorithm(initial_population=None,   pop_size=6,   generations=20,   crossover_rate=0.8,
mutation_rate=0.1):     if initial_population:
      population = initial_population[:pop_size]  # take only needed size
else:
      population = [random.randint(0, 31) for _ in range(pop_size)]
for gen in range(generations):
      population.sort(key=lambda x: fitness(x), reverse=True)
best = population[0]        print(f"Gen {gen}: Best x={best},
f(x)={fitness(best)}")        new_pop = [best]          while
len(new_pop) < pop_size:
         parent1 = tournament_selection(population)
parent2 = tournament_selection(population)            if
random.random() < crossover_rate:
            child1, child2 = crossover(parent1, parent2)
else:
            child1, child2 = parent1, parent2            child1 =
mutate(child1, mutation_rate)            child2 = mutate(child2,
mutation_rate)            new_pop.extend([child1, child2])        population
= new_pop[:pop_size]     population.sort(key=lambda x: fitness(x),
reverse=True)     best = population[0]     print(f"\nBest Solution:
x={best}, f(x)={fitness(best)}") custom_population = [3, 7, 15, 20, 25,
```

30] genetic_algorithm(initial_population=custom_population, generations=5)

Program 2 : Optimization via Gene expression

Problem statement:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning. **Algorithm:**

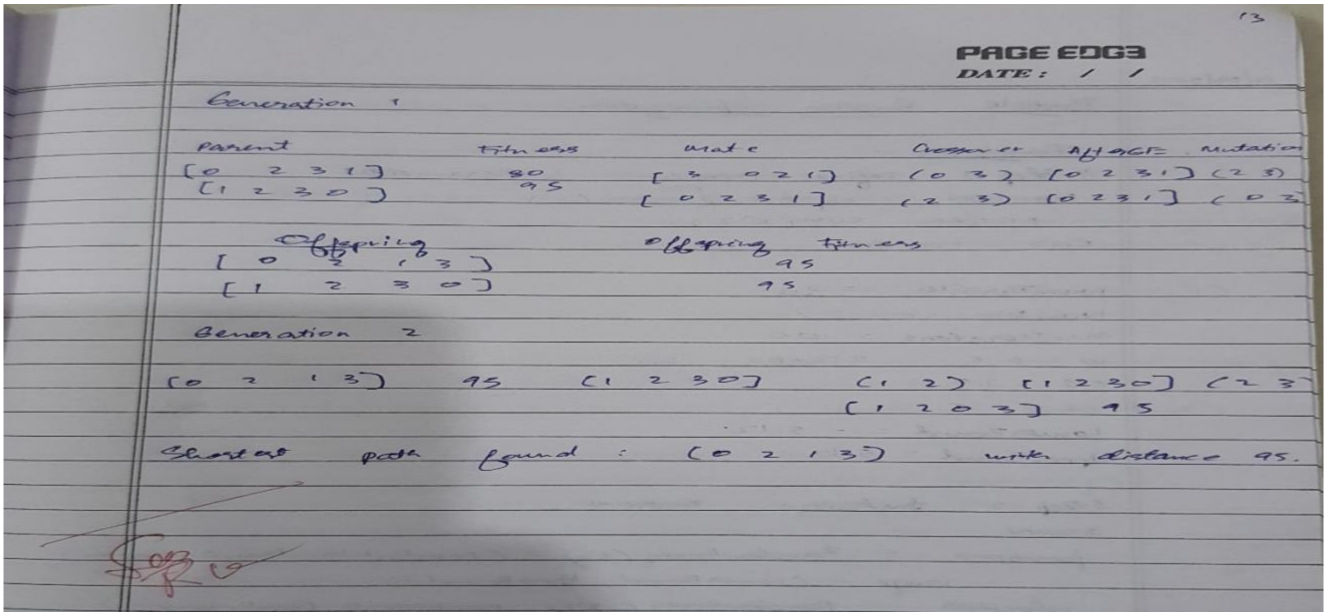
LAB-02. Optimization via Gene Expression Algorithm

1. Input : distance matrix, population size, generations
2. Initialize population with random gene sequences
3. best_path = None
   best_distance = Infinity
4. For each generation :
   For each pair of parents :
   - child = Crossover (parent1, parent2)
   - child = GeneExpression (child)
   - offspring = Mutate (child)
   - offspring = Gene Expression (offspring)
   - fitness = Calculate Distance (offspring)
   - If fitness < best distance :
     best_path = offspring
     best_distance = fitness
   Replace population with new offspring
5. Output best_path, best_distance .

Function GeneExpression (Sequence):
  Remove duplicates
  Add missing cities
  Return valid path

O/P Enter no of cities 4

0	10	15	20
10	0	35	25
15	35	0	30
20	25	30	0



13

PAGE EDGE
DATE: / /

Generation 1

Parent Fitness Mate Crossover After Mutation

[0 2 3 1] 80 [3 0 2 1] (0 3) [0 2 3 1] (2 3)

[1 2 3 0] 95 [0 2 3 1] (2 3) [0 2 3 1] (0 3

Offspring Offspring fitness

[0 2 1 3] 95

[1 2 3 0] 95

Generation 2

[0 2 1 3] 95 [1 2 3 0] (1 2) [1 2 3 0] (2 3

[1 2 0 3] 95

Shortest path found : [0 2 1 3] with distance 95.

Code:

```
import random
import math
POP_SIZE = 20
MUTATION_RATE = 0.1
GENERATIONS = 50
X_MIN, X_MAX = 0, 10
def fitness(x):
    return math.sin(x) * x

def initial_population():
    return [random.uniform(X_MIN, X_MAX) for _ in range(POP_SIZE)]


def select(population):
    contenders = random.sample(population, 3)
    return max(contenders, key=fitness)
```

```
def crossover(p1, p2):
    return (p1 + p2) / 2


def mutate(x):
    if random.random() < MUTATION_RATE:
        x += random.uniform(-0.5, 0.5)
        x = max(min(x, X_MAX), X_MIN)
    return x


def genetic_algorithm():
    population = initial_population()

    for generation in range(GENERATIONS):
        new_population = []

        for _ in range(POP_SIZE):
            parent1 = select(population)
            parent2 = select(population)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

        population = new_population
        best = max(population, key=fitness)
        print(f"Gen {generation}: Best x = {best:.4f}, f(x) = {fitness(best):.4f}")

    return best
best_solution = genetic_algorithm()
print("\nBest solution found:")
print(f"x = {best_solution:.4f}")
print(f"f(x) = {fitness(best_solution):.4f}")
```

<u>Program 3 : Particle swarm Optimization</u>

Problem statement:

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality.

Algorithm:

9/09/2025

Particle Swarm Optimization

Pseudocode :-

```
// Step 1 : Define Problem & Parameters
FUNCTION objective_function(x)        (PSO)
    RETURN SUM(x^2)
END FUNCTION

NumParticles = 30
NumDimensions = 2
MaxIterations = 100
w = 0.5        // Inertia We
c1 = 1.5
c2 = 1.5
LowerBound = - 5.12
UpperBound = 5.12

// Step 2 : Initialize Particles
BEGIN
positions = RandomArray(size = (NumParticles, NumDimensions),
        range = (LowerBound, UpperBound))
velocities = RandomArray( size = NumParticles, NumDimension),
  range = (-1, 1))
  // Initialize personal bests (pbest)
pbest_positions = COPY (positions)
pbest_fitness = [objective_function(p) FOR p IN
        pbest_positions]

// Initialize global
gbest_index = IndexOfMinimum( pbest_fitness)
gbest_position = pbest_Positions [ gbest_index]
 gbest_fitness = pbest_fitness [gbest_index]
```

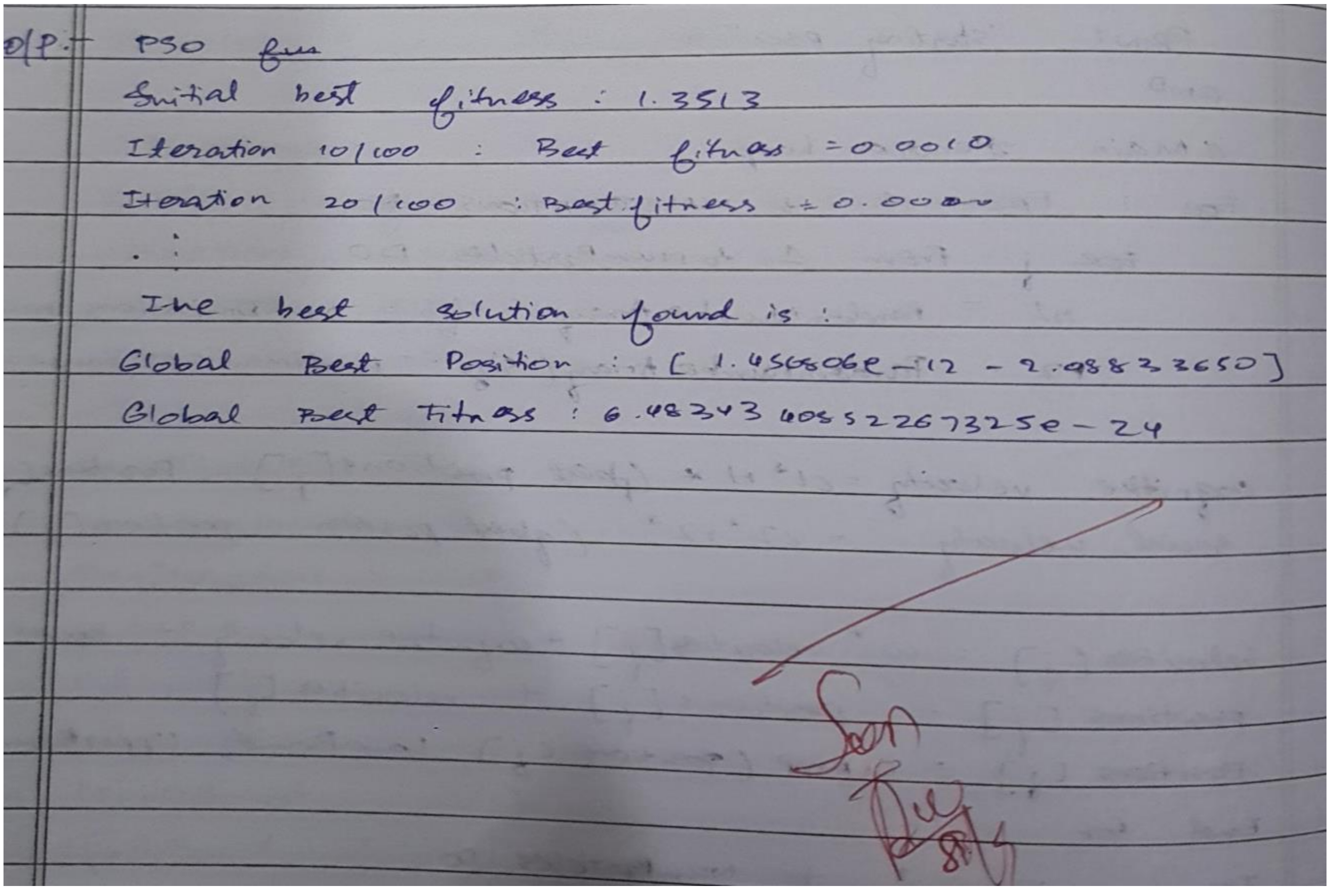
```
PRINT "starting PSO"
END

// Main Iteration Loop
FOR i FROM 1 to MAXITERATIONS DO
    FOR j From 1 to NumParticles DO
        r1 = RandomNumberArray( size = NumDimensions, range=(0,1))
        r2 = RandomNumberArray( size = NumDimensions, range=(0,1))

cognitive_velocity = c1* r1 * (pbest_positions[j] - Positions[j])
social_velocity = c2*r2* (gbest_position - positions[j])

velocities[j] = w* velocities[j] + cognitive_velocity + social_velocity
positions[j] = positions[j] + velocities[j]
positions[j] = CLAMP (positions[j], LowerBound, UpperBound)
End for
For j From 1 to NumParticles DO
    current_fitness = objective_function( positions[j])
    IF current_fitness < pbest_fitness[j] THEN
        pbest_fitness[j] = current_fitness
        pbest_positions[j] = positions[j]
    END IF
    IF current_fitness < gbest_fitness THEN
        gbest_fitness = current_fitness
        gbest_position = positions[j]
        END if
        END for

Return gbest_position
Return gbest_fitness
```



O/P:- PSO fun
Initial best fitness : 1.3513
Iteration 10/100 : Best fitness = 0.0010
Iteration 20/100 : Best fitness = 0.0000
⋮
The best solution found is :
Global Best Position : [1.456806e-12 -2.98833650]
Global Best Fitness : 6.48343 6085226 7325e-24

Code:

```
import random
import numpy as np
def fitness_function(position):
    return np.sum(position ** 2)

def PSO(dimensions, num_particles, max_iterations):
    w = 0.5
    c1 = 0.8
    c2 = 0.9
    swarm = []
    for _ in range(num_particles):
        position = np.random.uniform(-10, 10, dimensions)
        velocity = np.random.uniform(-1, 1, dimensions)
        pbest_position = position.copy()
        pbest_fitness = fitness_function(position)

        swarm.append({
            'position': position,
            'velocity': velocity,
```

```
            'pbest_position': pbest_position,
            'pbest_fitness': pbest_fitness
        })
    gbest_position = swarm[0]['pbest_position'].copy()
    gbest_fitness = fitness_function(gbest_position)

    for _ in range(max_iterations):
        for particle in swarm:
            fitness = fitness_function(particle['position'])
            if fitness < particle['pbest_fitness']:
                particle['pbest_fitness'] = fitness
                particle['pbest_position'] = particle['position'].copy()
            if fitness < gbest_fitness:
                gbest_fitness = fitness
                gbest_position = particle['position'].copy()
        for particle in swarm:
            rand1 = random.random()
            rand2 = random.random()

            inertia = w * particle['velocity']
            cognitive = c1 * rand1 * (particle['pbest_position'] - particle['position'])
            social = c2 * rand2 * (gbest_position - particle['position'])

            particle['velocity'] = inertia + cognitive + social
            particle['position'] = particle['position'] + particle['velocity']

    return gbest_position, gbest_fitness
best_position, best_fitness = PSO(dimensions=3, num_particles=30, max_iterations=100)
print("Best Position:", best_position)
print("Best Fitness:", best_fitness)
```

Program 4 : Ant Colony Optimization

Problem statement:

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:

08/09/2025

DATE: / /

LAB-04.

ANT COLONY OPTIMIZATION

Pseudocode.

```
Initialize parameters : m, alpha, beta, rho, Q, max_iter
Initialize pheromone levels τ_ij for all edges (i, j)
Set best_tour ← None
Set best_length ← infinity

FOR iter from 1 to max_iter :
    FOR each ant k from 1 to m :
        Place ant k on a random start city
        Initialize tabu list (visited cities) for ant k

        WHILE not all cities are visited :
            From current city i, select next city j
            with probability :
              P_ij = (τ_ij ^alpha * η_ij ^beta) / sum_over_allowed (P_ik)

        Move to city j and add to tabu list

    IF L_k < best_length :
        best_length ← L_k
        best_tour ← ant k's tour
    FOR each edge (i, j) :
        Evaporate pheromone :
            τ_ij ← (1 - rho) * τ_ij

    FOR each ant k :
        FOR each edge (i, j) in ant k's tour :
            Add pheromone :
```

$\tau\_ij \leftarrow \tau\_ij + (Q / L\_k)$

RETURN best_tour, best_length

Output :-

Best path found : [1, 3, 4, 2, 0]

Best path length : 22.35

Code:

```
import random
import math
cities = [(0, 0), (1, 5), (5, 2), (6, 6), (8, 3)]
num_cities = len(cities)
num_ants = 3
max_iter = 50

alpha = 1
beta = 2
evaporation = 0.5
pheromone_deposit = 100
def distance(a, b):
    x1, y1 = cities[a]
```

```
    x2, y2 = cities[b]
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

pheromone = [[1 for _ in range(num_cities)] for _ in range(num_cities)]

def choose_next_city(current_city, visited):
    probabilities = []
    for city in range(num_cities):
        if city in visited:
            probabilities.append(0)
        else:
            tau = pheromone[current_city][city] ** alpha
            eta = (1 / distance(current_city, city)) ** beta
            probabilities.append(tau * eta)
    total = sum(probabilities)
    if total == 0:
        return random.choice([c for c in range(num_cities) if c not in visited])
    probabilities = [p / total for p in probabilities]
    return random.choices(range(num_cities), weights=probabilities)[0]

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

best_path = None
best_length = float('inf')

for iteration in range(max_iter):
    all_paths = []
    for _ in range(num_ants):
        start = random.randint(0, num_cities - 1)
        path = [start]
        visited = set(path)
        while len(path) < num_cities:
            next_city = choose_next_city(path[-1], visited)
            path.append(next_city)
```

```
            visited.add(next_city)
        length = path_length(path)
        all_paths.append((path, length))
        if length < best_length:
            best_length = length
            best_path = path

    for i in range(num_cities):
        for j in range(num_cities):
            pheromone[i][j] *= (1 - evaporation)

    for path, length in all_paths:
        for i in range(num_cities):
            from_city = path[i]
            to_city = path[(i + 1) % num_cities]
            pheromone[from_city][to_city] += pheromone_deposit / length

print("Best path found:", best_path)
print("Best path length:", round(best_length, 2))
```

Program 5 : Cuckoo search Optimization

Problem statement:

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:

15/09/2025

DATE: / /

LAB - 05

CUCKOO SEARCH ALGORITHM

Input:

n : Initial population size (number of nests)

P_a : Fraction of worse nests to be abandoned and replaced.

Max iterations: Max. number of iterations

$f(x)$: Objective function to optimize

Output :-

Best Nest (Solution) found

Step 1: Initialization

1. Set initial value of the host nest size n, probability $P_a \in (0,1)$ and Maximum number of iterations Max_t.

2. Set iteration counter t : 0

3. For i = 1 to n :

→ Generate initial population of n host nests x_i^t

→ Evaluate fitness function $f(x_i^t)$

Step 2 : Generalization

1. Generate a new solution (cuckoo) x_i^{t+1} randomly by Levi flight

2. Evaluate fitness function $f(x_i^{t+1})$

Step 3 : Selection (Replace Worst Nests)

1. Randomly chose a nest x_j from the population.

2. If $f(x_i^{t+1}) > f(x_j^t)$

3. Replace nest x_j with the new solution x_i^{t+1}

Step 4 : Abandonment & Replacement

1. Abandon a fraction P_a of worst nests .
2. Build new nests using Levy flight
3. Keep best solution

Step 5 : Ranking & Update .

1. Rank all nests based on fitness
2. Find current best solution
3. Increment iteration counter : $t = t + 1$

Step 7 : Termination .

1. Repeat steps until $t \geq Max_t$
2. Produce and return best solution found

__Output :-__

Number of cities : 4
City coordinates :
City 0 : [1 1]
City 1 : [6 5]
City 2 : [7 2]
City 3 : [3 8]

Best route :
2 → 0 → 3 → 1 → 2

Fitness (Total distance) : 20.7678

Code:

```
import numpy as np
import random


cities = np.array([
    [1, 1], [4, 5], [7, 2], [3, 8]
])
n = len(cities)
def distance_matrix(coords):
    return np.linalg.norm(coords[:, None] - coords, axis=2)
dist = distance_matrix(cities)
def route_dist(route):
    d = sum(dist[route[i], route[i+1]] for i in range(n-1))
    d += dist[route[-1], route[0]]  # return to start
    return d
def swap(route):
    r = route.copy()
    i, j = random.sample(range(n), 2)
    r[i], r[j] = r[j], r[i]
    return r

def cuckoo_search(nests=10, Pa=0.25, max_iter=100):
    population = [np.random.permutation(n) for _ in range(nests)]
    fitness = [route_dist(p) for p in population]

    best_idx = np.argmin(fitness)
    best_route, best_fit = population[best_idx], fitness[best_idx]

    for _ in range(max_iter):
        for i in range(nests):
            new_sol = swap(population[i])
            new_fit = route_dist(new_sol)
            j = random.randint(0, nests - 1)
            if new_fit < fitness[j]:
                population[j], fitness[j] = new_sol, new_fit
                if new_fit < best_fit:
```

```
                best_route, best_fit = new_sol, new_fit
        worst_count = int(Pa * nests)
        worst_idxs = np.argsort(fitness)[-worst_count:]
        for idx in worst_idxs:
            population[idx] = np.random.permutation(n)
            fitness[idx] = route_dist(population[idx])
            if fitness[idx] < best_fit:
                best_route, best_fit = population[idx], fitness[idx]

    return best_route, best_fit
best_route, best_fitness = cuckoo_search()
print(f"Number of cities: {n}")
print("City coordinates:")
for i, c in enumerate(cities):
    print(f"City {i}: {c}")
print("\nBest route:")
print(" -> ".join(map(str, best_route)) + f" -> {best_route[0]}")
print(f"Fitness (Total distance): {best_fitness:.4f}")
```

Program 6 : Grey Wolf Optimization

Problem statement:

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:

29/09/2025

DATE: / /

GREY WOLF OPTIMIZATION (GWO)

~~Pseudocode~~. Algorithm.

Step 1: Initialization

1\. Define Parameters: Set the problem dimensions (dim), search bounds (bounds), maximum number of iterations (max-iterations), and the size of the wolf pack (num_wolves)

2\. def GWO_Algorithm_Sketch (objective_func, dim, bounds, max_iterations, num_wolves):
    for iteration in range(max_iterations):
        for i in range(num_wolves):
            elif fitness < Delta_score:

def GWO_Algorithm_Sketch(objective_func, dim, bounds, max_iterations, num_wolves):

wolf-positions = random_initialization (num_wolves, dim, bounds)

Alpha_pos, Beta_pos, Delta_pos = zeros(dim), zeros(dim), zeros(dim)

Alpha_score = Beta_score = Delta_score = infinity

for iteration in range(max_iterations):
    for i in range(num_wolves):
        fitness = objective_func(wolf_positions[i])

if fitness < Alpha_score:
    update_leaders(wolf_positions[i], fitness, 'Alpha')
elif fitness < Beta_score:

```
update_leaders ( wolf_positions [i], fitness, 'Beta')
elif fitness < Delta_score:
update_leaders ( wolf_positions [i], fitness, 'Delta')

a = 2 - iteration * (2 / max_iterations)

for i in range( num_wolves ):
    current_pos = wolf_positions[i]

x1 = calculate_component (Alpha_pos, current_pos, a, 'A1', 'C1')
x2 = Calculate_component ( Beta_pos, current_pos, a, 'A2', 'C2')
x3 = Calculate_component ( Delta_pos, current_pos, a, 'A3', 'C3')
new_pos = (x1 + x2 + x3) / 3
wolf_positions[i] = clip_to_bounds ( new_pos, bounds)

return Alpha_pos, Alpha_score

def calculate_component ( leader_pos, current_pos, a, A_var, C_var)
r1, r2 = random_vector(), random_vector()
A_var = 2* a* r2 - a
C_var = 2* r2
D_leader = absolute_value( C_var* leader_pos - current_pos)
x_component = leader_pos - A_var * D_leader
return x_component
```

(Implementation) o/p for TSP:-

Number of Cities : 5
Best Route found : [0 3 2 4 1]
Minimum Tour Distance : 25.2604

Code:

```
import numpy as np

def distance(path, dist_matrix):
    total = 0
    for i in range(len(path)):
        total +=
dist_matrix[path[i]][path[(i+1)%len(path)
]]
    return total

def vector_to_path(vec):
    return np.argsort(vec)

def GWO_TSP(dist_matrix,
num_wolves=20, max_iter=200):
    num_cities = len(dist_matrix)
    wolves = np.random.rand(num_wolves,
num_cities)

    alpha = beta = delta = None
    alpha_cost = beta_cost = delta_cost =
float("inf")

    for t in range(max_iter):
        for w in wolves:
            path = vector_to_path(w)
            cost = distance(path, dist_matrix)

            if cost < alpha_cost:
                delta_cost, delta = beta_cost,
beta
                beta_cost, beta = alpha_cost,
alpha
                alpha_cost, alpha = cost,
w.copy()
            elif cost < beta_cost:
                delta_cost, delta = beta_cost,
beta
                beta_cost, beta = cost, w.copy()
            elif cost < delta_cost:
                delta_cost, delta = cost,
w.copy()

        a = 2 - 2 * (t / max_iter)
        for i in range(num_wolves):
            for j in range(num_cities):
```

```
            r1, r2 = np.random.rand(),
np.random.rand()
            A1 = 2*a*r1 - a
            C1 = 2*r2
            D_alpha = abs(C1 * alpha[j] -
wolves[i][j])
            X1 = alpha[j] - A1 * D_alpha

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

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

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

    best_path = vector_to_path(alpha)
    return best_path, alpha_cost
dist_matrix = [
    [0, 10, 12, 11],
    [10, 0, 13, 5],
    [12, 13, 0, 9],
    [11, 5, 9, 0]
]

best_route, best_distance =
GWO_TSP(dist_matrix, num_wolves=20,
max_iter=200)
print("Best Route Found:", best_route)
print("Best Route Distance:",
best_distance)
```

Program 7 : Parallel cellular Optimization

Problem statement:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to

PAGE EDGE

DATE : / /

13/11/2025

LAB - 07

Parallel cellular Algorithm.

1. Define the Problem :- Choose what function to optimize
2. Set Parameters : Grid size (20x20), max iterations (1000)
3. Create Random Cells : Spread cells randomly across search space.
4. Evaluate Fitness : Test how good each cell's Position is
5. Update Cells : Each cell looks at neighbors and moves toward better ones.
6. Repeat : Keep updating for many generations
7. Return Best : Output the best solution found.

Pseudocode :-

```
import numpy as np

def simple_ca_optimize ( func, bounds, size = 0, iters = 50):
    dim = len(bounds)
    cells = np.random.rand( size, size, dim)
    for d in range(dim) :
        min_b, max_b = bounds [d]
        cells [:, :, d] = cells[:, :, d] * (max_b - min_b)
                                          + min_b
```

explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Application → Complex math functions (Sphere

```
best_pos = None
best_val = float('inf')

for step in range(iters):
    for i in range(size):
        for j in range(size):
            val = func(cells[i, j])
            if val < best_val:
                best_val = val
                best_pos = cells[i, j].copy()

            best_neighbour = None
            best_neighbour_val = float('inf')
            for di, dj in [(-1,0), (1,0), (0,-1), (0,1)]:
                ni, nj = (i+di) % size, (j+dj) % size
                nval = func(cells[ni, nj])
                if nval < best_neighbour_val:
                    best_neighbour_val = nval
                    best_neighbour = cells[ni, nj]

            if best_neighbour_val < val:
                cells[i, j] = 0.5 * (cells[i, j] + best_neighbour)
return best_pos, best_val

def sphere(x):
    return x[0]**2 + x[1]**2
```

} Sphere function

find minimum at (0,0)

Low

Code:

```
import numpy as np

def simple_ao(func, bounds, size=10, iters=50):
    dim = len(bounds)
    cells = np.random.rand(size, size, dim)
    for d in range(dim):
        min_b, max_b = bounds[d]
        cells[:, :, d] = cells[:, :, d] * (max_b - min_b) + min_b

    best_pos = None
    best_val = float('inf')

    for step in range(iters):
        for i in range(size):
            for j in range(size):
                val = func(cells[i, j])
                if val < best_val:
                    best_val = val
                    best_pos = cells[i, j].copy()
        new_cells = cells.copy()
        for i in range(size):
            for j in range(size):
                best_neighbor = cells[i, j]
                best_neighbor_val = func(best_neighbor)

                for di, dj in [(-1,0), (1,0), (0,-1), (0,1)]:
                    ni = (i + di) % size
                    nj = (j + dj) % size
                    nval = func(cells[ni, nj])
                    if nval < best_neighbor_val:
                        best_neighbor_val = nval
                        best_neighbor = cells[ni, nj]
                new_cells[i, j] = 0.5 * (cells[i, j] + best_neighbor)

        cells = new_cells

    return best_pos, best_val
def sphere(x):
    return np.sum(x**2)
best_pos, best_val = simple_ao(
    func=sphere,
    bounds=[(-10, 10), (-10, 10)],
    size=10,
    iters=50
```

```
)

print("Best Position:", best_pos)
print("Best Value:", best_val)
```