VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Chitrashree K (1BM23CS081)

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 **Chitrashree K (1BM23CS081),** 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.

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

Index

Github Link:
https://github.com/Chitrashree-tech/Bio_Inspired_Systems

Program 1

Genetic Algorithm for Optimization Problems:

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. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:

Bafna Gold

18/8/25

LAB - 01 ① Genetic Algorithm for Optimization Problem

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

1.] 1. Roulette wheel selection :- probability of selecting an individual is proportional to its fitness.

2. Tournament selection :-

Ex :- $x \rightarrow 0 - 31$

2]

Initial pop	x val	func $f(x) = x^2$	prob	% prob	expected output	actual count
01100	12	144	0.1247	12.47	0.49	1
11001	25	625	0.5411	54.11	2.16	2
00101	5	25	0.0216	2.16	0.08	0
10011	19	361	0.3126	31.26	1.25	1
		1155	1.0	100	4	
		288.75	0.25	25	1	
		625	0.5411	54.11	2.16	

$prob = \frac{f(x)}{\sum f(x)}$; expected output : $\frac{f(x)}{avg(\sum f(x))}$

3] → crossover point is chosen randomly

Mating pool	crossover pt	offspring after crossover	x val	func $f(x) = x^2$
01|100	4	01101	13	169
11|001		11000	24	576
11|001	3	11101	29	729
10|011		10011	19	289
				1763
				440.75
				729

→ chosen randomly

Mutation

string no	offspring after crossover	mutation chromosome	offspring after mutation	x val	func $f(x) = x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
					2546
					636.5
					841

⇒ Applications:-

1. Travelling salesman problem → sequence of cities - minimize travel distance
2. Job scheduling → sequence of job to machine assignments - (Min) negative of makespan
3. Feature selection → 1 - select ; 0 - ignore - (Max) model accuracy.
4. Knapsack problem → 1 - include, 0 - exclude - (max) value

⇒ Implementation :- Knapsack Problem :-

1] problem capacity 15 - kg

item	1	2	3	4	5	6
weight	2	5	10	8	3	7
value	6	10	18	12	7	14

2] chromosome 1 - include ; 0 - exclude

3] initial population - 010101 , 111000, 001111, 101001

4] fitness eval - 101001 - 19 - 0
010101 - 20 - 0 None valid
111000 - 27 - 0
001111 - 28 - 0

Bafna Gold

	mating pool	crossover pt	offspring after crossover	x val	fun of eqn f(x)	weight <15
not needed 5	101001	- mutated	- 100001		valid	
	010101	- mutated	- 010100		valid	
	111000	- unchanged	- invalid			
	001111	- mutated	- 001101		invalid	

max - 22

new population - 100001, 010100, 111000, 001101

100001 - 20 - valid

chromosome	items	weight	value	fitness
100001	1, 6	9	20	20
010100	2, 4	13	22	22
111000	1, 2, 3	17	34	0
00110	3, 5, 6	20	39	0

continues

⇒ Pseudo code:-

Input: n = no of items
W = maximum capacity
pop-size = population size
max-gen = max no of generations
pc = crossover prob
pm = mutation prob

Output: Best sol^n found

1. Initialize population
- Generate pop-size random chromosomes (sol^n)

2. Evaluate population
- Compute fitness of each chromosomes

3. Repeat until generation = max_gen:
   a) Selection
      - select parents from population based on fitness (eg. roulette wheel, tournament)
   b) Crossover
      - with probability pc:
        perform crossover the selected parents
      - otherwise, copy parents directly
   c) Mutation
      - For each offspring chromosome:
        - For each gene:
          - with probability pm, flip/alter the gene
   d) Evaluate offspring
      - Compute fitness of each new chromosome
   e) Replacement
      - Form the new population (offspring + may elitism)
   f) Update best solution found so far

4. Return the best solution and its fitness.

Input:- (1, 2, 6), (2, 5, 10), (3, 10, 18), (4, 8, 12), (5, 3, 7), (6, 7, 14)

Output:- Gen 1:- Best fitness = 27, chromosome = [1, 1, 0, 0, 1, 0]
Gen 2:- ——— = 31, ——— = [1, 1, 0, 0, 0, 1]
Gen 3:- ——— = 31, ——— = [1, 1, 0, 0, 0, 1]

Gen 20:- ——— = 39, chromosome = [0, 1, 1, 0, 0, 1]
Best Sol^n:- chromosome [0, 1, 1, 0, 0, 1]
Fitness val = 39, weight = 15, items: [2, 3, 6]

Code:

```
import random

# Items: (weight, value)
items = [
    (2, 6),   # Item1
    (5, 10),  # Item2
    (10, 18), # Item3
    (8, 12),  # Item4
    (3, 7),   # Item5
    (7, 14)   # Item6
]
capacity = 15

# Parameters
POP_SIZE = 6
GENS = 20
CROSS_RATE = 0.8
MUT_RATE = 0.1

# Generate initial population
def init_population():
    return [[random.randint(0, 1) for _ in range(len(items))] for _ in range(POP_SIZE)]

# Fitness function
def fitness(chromosome):
    total_weight, total_value = 0, 0
    for gene, (w, v) in zip(chromosome, items):
        if gene == 1:
            total_weight += w
            total_value += v
    return total_value if total_weight <= capacity else 0

# Roulette Wheel Selection
def selection(pop, fits):
    total_fit = sum(fits)
    if total_fit == 0:
        return random.choice(pop)
    pick = random.uniform(0, total_fit)
    current = 0
    for chromosome, fit in zip(pop, fits):
        current += fit
        if current > pick:
            return chromosome

# Crossover (single point)
def crossover(p1, p2):
    if random.random() < CROSS_RATE:
```

```
        point = random.randint(1, len(p1)-1)
        return p1[:point] + p2[point:], p2[:point] + p1[point:]
    return p1, p2

# Mutation (bit flip)
def mutate(chromosome):
    return [1-g if random.random() < MUT_RATE else g for g in chromosome]

# Run GA
def genetic_algorithm():
    population = init_population()

    for gen in range(GENS):
        fits = [fitness(ch) for ch in population]
        new_pop = []

        for _ in range(POP_SIZE // 2):
            p1, p2 = selection(population, fits), selection(population, fits)
            c1, c2 = crossover(p1, p2)
            new_pop.extend([mutate(c1), mutate(c2)])

        population = new_pop
        best_fit = max(fits)
        best_ch = population[fits.index(best_fit)]
        print(f"Gen {gen+1}: Best fitness = {best_fit}, Chromosome = {best_ch}")

    # Final best
    final_fits = [fitness(ch) for ch in population]
    best_fit = max(final_fits)
    best_ch = population[final_fits.index(best_fit)]
    print("\nBest Solution:")
    print("Chromosome:", best_ch)
    print("Fitness (Value):", best_fit)
    print("Weight:", sum(w for g,(w,_) in zip(best_ch, items) if g==1))
    print("Items:", [i+1 for i,g in enumerate(best_ch) if g==1])

# Run
if __name__ == "__main__":
    genetic_algorithm()
```

Output:

```
Gen 1: Best fitness = 30, Chromosome = [1, 1, 0, 0, 1, 0]
Gen 2: Best fitness = 24, Chromosome = [1, 1, 0, 0, 1, 0]
Gen 3: Best fitness = 23, Chromosome = [0, 1, 0, 1, 1, 1]
Gen 4: Best fitness = 30, Chromosome = [0, 1, 1, 0, 1, 0]
Gen 5: Best fitness = 27, Chromosome = [0, 1, 0, 0, 1, 0]
Gen 6: Best fitness = 27, Chromosome = [0, 0, 0, 1, 1, 1]
Gen 7: Best fitness = 25, Chromosome = [0, 0, 1, 1, 0, 0]
Gen 8: Best fitness = 28, Chromosome = [1, 1, 0, 0, 1, 0]
Gen 9: Best fitness = 25, Chromosome = [0, 1, 0, 0, 1, 0]
Gen 10: Best fitness = 23, Chromosome = [1, 1, 0, 0, 0, 0]
Gen 11: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 12: Best fitness = 31, Chromosome = [1, 0, 0, 0, 1, 0]
Gen 13: Best fitness = 31, Chromosome = [1, 1, 1, 0, 1, 0]
Gen 14: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 15: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 16: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 17: Best fitness = 31, Chromosome = [1, 0, 1, 0, 0, 0]
Gen 18: Best fitness = 31, Chromosome = [1, 0, 1, 0, 0, 0]
Gen 19: Best fitness = 31, Chromosome = [1, 1, 1, 0, 1, 0]
Gen 20: Best fitness = 31, Chromosome = [0, 0, 1, 0, 0, 0]

Best Solution:
Chromosome: [1, 0, 1, 0, 1, 0]
Fitness (Value): 31
Weight: 15
Items: [1, 3, 5]

=== Code Execution Successful ===
```

Program 2:

Optimization via Gene Expression Algorithms:

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:

25/8/25

LAB-12 ⑦ Optimization via Gene Expression Algorithm

Steps: Initialize population
Evaluate fitness
Selection, crossover, mutation
Gene expression, iterate

Ex: for $f(x) = x^2$ Gene expressn each gene sequence into a mathematical expressn

Sl No	Initial popu	x	f(x)	prob	expected cnt	actual count
C1	01010	10	100	0.072	0.29	0
C2	11100	28	784	0.57	2.28	2
C3	10101	21	441	0.32	1.28	1
C4	00111	7	49	0.03	0.14	0
			1374	1.0		
			343.5	0.25		
			784	0.57		

Cross over	Mating pool	cross over	offspring after crossover	x val	func^n
	11100	11101	11101	29	841
	10101	10100	10100	20	400
					841

Mutation: 11101 - no mutation
10100 - last bit flipped - 10101

Pseudo-Code:-

1. Initialize parameters:
- pop-size - no of chromosomes
- ch_len - len of each gene sequence
- func^n: set of operators
- terminals - set of numbers
- crossover_rate, mutation_rate, generations

2. Generate initial population:-
for each chromosome:
Randomly create a sequence of genes

3. for gen = 1 to generations:-
a) Decode chrome to expressⁿ → evaluate x
b) compute fitness f(x)
c) Select parents using roulette wheel based on fitness
d) crossover to produce children
e) Apply mutation to children
f) form new population from offspring
g) track best solⁿ of this gen

4. Return the best chromosome, expressⁿ, x, fit

Applications:- Mathematical optimization
- encoding arithmetic expressⁿ

Output: Gen 1: Best = 7-*+68T, f(x) = D
Gen 2: Best = 765+5+-, f(x) = D
Gen 3: Best = 5+9+6-7, f(x) = 169
:
Gen 20: Best = 5+9+9-1, f(x) = 484

Best solⁿ: Chromosome: 5+9+9-1
Fitness: 484

26/8/2025

Code:

```
import random
import math

# --------------------------
# PARAMETERS
# --------------------------
POP_SIZE = 6
CHROM_LENGTH = 7        # length of genetic sequence
FUNCTIONS = ['+', '-', '*']
TERMINALS = [str(i) for i in range(10)]  # constants 0–9
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.1
GENERATIONS = 20

# --------------------------
# HELPER FUNCTIONS
# --------------------------

def random_gene():
    """Return a random gene (either function or terminal)."""
    if random.random() < 0.4:
        return random.choice(FUNCTIONS)
    return random.choice(TERMINALS)

def create_individual():
    """Generate a random chromosome (sequence)."""
    return [random_gene() for _ in range(CHROM_LENGTH)]

def decode_expression(chromosome):
    """Convert chromosome into a valid arithmetic expression."""
    expr = ""
    for gene in chromosome:
        expr += gene
    return expr

def evaluate(chromosome):
    """Evaluate chromosome by expressing it as integer x, then f(x)=x^2."""
    expr = decode_expression(chromosome)
    try:
        # Evaluate safely
        x_val = int(eval(expr))
    except Exception:
        return 0  # invalid expression
    if x_val < 0 or x_val > 31:  # constrain to problem domain
        return 0
    return x_val**2
```

```
def roulette_wheel_selection(pop, fitnesses):
    """Select one individual using roulette wheel."""
    total_fit = sum(fitnesses)
    if total_fit == 0:
        return random.choice(pop)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return pop[i]

def crossover(parent1, parent2):
    """Single point crossover."""
    if random.random() > CROSSOVER_RATE:
        return parent1[:], parent2[:]
    point = random.randint(1, CHROM_LENGTH - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def mutate(chromosome):
    """Mutate chromosome by flipping a gene."""
    for i in range(len(chromosome)):
        if random.random() < MUTATION_RATE:
            chromosome[i] = random_gene()
    return chromosome

# ---------------------------
# MAIN LOOP
# ---------------------------

population = [create_individual() for _ in range(POP_SIZE)]

for gen in range(GENERATIONS):
    fitnesses = [evaluate(ind) for ind in population]
    best_index = fitnesses.index(max(fitnesses))
    best = population[best_index]
    print(f"Gen {gen+1}: Best = {decode_expression(best)}, f(x) = {max(fitnesses)}")

    # New population
    new_population = []
    while len(new_population) < POP_SIZE:
        p1 = roulette_wheel_selection(population, fitnesses)
        p2 = roulette_wheel_selection(population, fitnesses)
        c1, c2 = crossover(p1, p2)
        c1 = mutate(c1)
        c2 = mutate(c2)
```

```
        new_population.extend([c1, c2])
    population = new_population[:POP_SIZE]

# Final result
fitnesses = [evaluate(ind) for ind in population]
best_index = fitnesses.index(max(fitnesses))
best = population[best_index]
print("\nFinal Best Solution:")
print("Chromosome:", decode_expression(best))
print("Fitness:", max(fitnesses))
```

Output:

```
Gen 1: Best = 2*++9++, f(x) = 0
Gen 2: Best = 8**2+8+, f(x) = 0
Gen 3: Best = 2*++3+3, f(x) = 81
Gen 4: Best = 7*++3+3, f(x) = 576
Gen 5: Best = 7*++3+3, f(x) = 576
Gen 6: Best = 9*++3+3, f(x) = 900
Gen 7: Best = 9*++3+3, f(x) = 900
Gen 8: Best = 8*++3+3, f(x) = 729
Gen 9: Best = 8*++3+3, f(x) = 729
Gen 10: Best = 8*++3+3, f(x) = 729
Gen 11: Best = 8*++3+3, f(x) = 729
Gen 12: Best = 8*++3+3, f(x) = 729
Gen 13: Best = 8*++3+3, f(x) = 729
Gen 14: Best = 6*++3+3, f(x) = 441
Gen 15: Best = 6*++3+3, f(x) = 441
Gen 16: Best = 6*4+1+3, f(x) = 784
Gen 17: Best = 6*4+1+3, f(x) = 784
Gen 18: Best = 6*4+1+3, f(x) = 784
Gen 19: Best = 6*4+3+3, f(x) = 900
Gen 20: Best = 6*4+3+3, f(x) = 900

Final Best Solution:
Chromosome: 6*4+3+3
Fitness: 900

=== Code Execution Successful ===
```

Program 3:

Particle Swarm Optimization for Function Optimization:

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. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

inertia → 0.4 - 0.9
c1 → 1.5 - 2.0
c2 → 1.5 - 2.0
R → 0 - 1

Bafna Gold
Date: Page:

21/09/2025
LAB - 03

② Particle Swarm Optimization for Function Optimization

Algorithm:-

Step1:- Create a population of agents, initialize swarm
Step2:- evaluate the fitness and update the personal best and the global best, velocity & position
Step3:- Keep updating until it has reached the maximum iterations or convergence

Pseudo code:-

(func^n)
$f(x) = x^2 + y^2$
no-particles = 10, w = 0.5, c1 = c2 = 1.5
iterations = 20

positions = random (no-particles)
velocities = random (no-particles)
pbest = positions
pbest-val = fitness-func^n (positions)
gbest = best (pbest_val, positions)

for t in range (iterations):
    for i in range (no-particles):
        velocities [i] = w * velocities [i] + c1 * rand() * (pbest[i] - positions[i]) + c2 * rand() * (gbest - positions [i])
        positions[i] += velocities [i]

        if fitness (position [i] > pbest_val[i]):
            pbest [i] = positions [i]
            pbest_val [i] = fitness (position [i])
    gbest = best (pbest_val, pbest)
print (gbest, f(gbest))

formulae :-

1. Updating velocity.
$v_i = w v_i + c_1 r_1 (pbest_i - x_i) + c_2 r_2 (gbest - x_i)$
2. updating position :- $x_i = x_i + v_i$

- c1 - how strongly the particle trusts its own past experience.
- r1 - adds randomness, so it doesn't always fly straight to pbest but instead explores around it.

- Applications :-
1. Robotics (drones) and path planning
2. Scheduling and Resource Allocation

c1 - personal past success, c2 - best soln found by group

- Drone :- to reach target, avoid obstacle, save energy / time, stay safe.

Fitness = α . pathlen + β . obstacle penalty + γ . energy used.

ex :- Path A - short but goes near a building → high penalty, so fitness is worse
Path B - bit longer but safe & smooth - better fitness score.
Path C - zig-zags a lot (wasting energy) - worse than B

Output :-
- inputs :- $f(x) = x^2 + x^3$
gens - 20 w = 0.5 c1 = c2 = 1.5
Output :- Global Best Soln :- 125.1671808
Fitness Value :- 1976638.9124...

input - $f(x) = x^2 + 3x + 9$
$w = 0.4$ $c1 = 1.5$ $c2 = 1$
Output :- Global Best Soln = 56.88275
Fitness Value : 3410.2964

Application :- Drone path planning
$c1 >> c2$ - Individualistic - good for exploration but coordination fails
$c2 >> c1$ - fast convergence

input :- n-drones = 3
Iterations = 30
$w = 0.5$
$c1 = c2 = 2.0$
goal - [10, 10]
fitness func^n - $\sqrt{(x-x_g)^2 + (y-y_g)^2}$

Output :- Final global Best Position = [8.96 9.02]
Final global Best Score = 1.305
$\sqrt{(8.96-10)^2 + (9.02-10)^2}$

Code:

```
import random

# Step 1: Define function (to maximize)
def f(x):
    return (x**2+3*x+4)   # simple quadratic

# Step 2: Parameters
num_particles = 10
iterations = 20
w = 0.4# inertia
c1, c2 = 1.5, 1

# Step 3: Initialize particles
positions = [random.uniform(-10, 10) for _ in range(num_particles)]
velocities = [random.uniform(-1, 1) for _ in range(num_particles)]
pbest = positions[:]
pbest_val = [f(x) for x in positions]
gbest = pbest[pbest_val.index(max(pbest_val))]

# Step 4–6: Iterate
for _ in range(iterations):
    for i in range(num_particles):
        # Update velocity
        velocities[i] = (w*velocities[i]
                         + c1*random.random()*(pbest[i]-positions[i])
                         + c2*random.random()*(gbest-positions[i]))
        # Update position
        positions[i] += velocities[i]

        # Update personal best
        val = f(positions[i])
        if val > pbest_val[i]:
            pbest[i] = positions[i]
            pbest_val[i] = val

    # Update global best
    gbest = pbest[pbest_val.index(max(pbest_val))]

# Step 7: Output
print("Global Best solution:", gbest, "Fitness Value:", f(gbest))
```

Output:

Output

```
Global Best solution: 26.806206878454258 Fitness Value: 802.9913478458512

=== Code Execution Successful ===
```

Program 4:

Ant Colony Optimization for the Traveling Salesman Problem:

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:

08/07/2025

LAB-04 ⑤ Ant Colony Optimization

Algorithm:-

step-01: Initialize pheromone trails and parameters

step-2: Each ant builds a tour using pheromone and distance based probabilities.

step-3: Evaluate all tours and update the best solution

step 4: Evaporate old pheromones and deposit new ones based on tour quality.

step 5: Repeat steps 2-4 until stopping condition is met.

step 6: Output the best tour and its length.

Pseudo Code:-

```
- For iter = 1 to iterations do:
    all_tours ← []
    For each ant in num_ants do:
        start_city ← random_city
        tour ← [start_city]
        unvisited ← all_cities - {start_city}
        while unvisited not empty do:
            For each city j in unvisited:
                probability[j] ← (pheromone[current][j]^alpha) * (1/distance[current][j]^beta)
            Normalize probabilities
            next_city ← select city using probability distribution
            tour.append(next_city)
            unvisited ← unvisited - {next_city}
```

Close the tour (return to start)
all_tours.append(tour)

length ← compute_tour_length(tour)
if length < best_length:
    best_length ← length
    best_tour ← tour

Update pheromones:
evaporate: pheromone ← pheromone * (1 - ρ)
for each tour in all_tours:
    for each edge (i, j) in tour:
        pheromone[i][j] ← pheromone[i][j]
        + Q / tour_length

input: no_ants = 10
iterations = 50
alpha = 1.0 → pheromone importance
beta = 5.0 → heuristic importance → high - prefer nearest cities
rho = 0.5 → evaporation rate
Q = 100 pheromone deposit factor
↓ how much pheromone is laid
large Q → strong reinforcement of good tours
small Q → weaker → slower convergence

high → ants strongly follow pheromone ones
low → rely less on pheromone
less depend more on pheromone than distance

output: It 1: Best length: 2235103, Best tour: [3, 4, 2, 0, 1]
α = 1
β = 3-5
ρ = 0.3 - 0.7
Q = 100 × 20.0
It: 50: Best length - 22.35 1032, Best tour [3, 4, 2, 1, 1]
Final Best Tour: [3, 4, 2, 0, 1]
Final Best Length - 22.3510

- Applications :-
1. Travelling salesmen problem.
2. Assignment problem
3. Network Routing : to find the best path for better communication
   - α - history of successful routes
   β - current quality of the link (bandwidth, delay, hop count)
   δ - routing memory fades
   a - good path is reinforced when packets reach the destination

* pheromone - value / metric stored in routers / nodes

QoS metrics :-

MG 8/9/23

Code:

```
import random
import math

# -------------------------
# Problem Setup (TSP Cities)
# -------------------------
cities = {
    0: (0, 0),
    1: (1, 5),
    2: (5, 2),
    3: (6, 6),
    4: (8, 3)
}

num_cities = len(cities)

# Distance matrix
distances = [[0] * num_cities for _ in range(num_cities)]
for i in range(num_cities):
    for j in range(num_cities):
        xi, yi = cities[i]
        xj, yj = cities[j]
        distances[i][j] = math.sqrt((xi - xj) ** 2 + (yi - yj) ** 2)

# -------------------------
# Parameters
# -------------------------
num_ants = 10
iterations = 50
alpha = 0      # pheromone importance
beta = 5.0       # heuristic importance
rho = 0.5       # evaporation rate
Q = 5       # pheromone deposit factor
initial_pheromone = 1.0

# -------------------------
# Initialize pheromones
# -------------------------
pheromone = [[initial_pheromone] * num_cities for _ in range(num_cities)]

# -------------------------
# Helper Functions
# -------------------------
def tour_length(tour):
    length = 0
    for i in range(len(tour) - 1):
```

```
        length += distances[tour[i]][tour[i + 1]]
    length += distances[tour[-1]][tour[0]]  # return to start
    return length

def select_next_city(current, unvisited):
    probabilities = []
    denom = sum((pheromone[current][j] ** alpha) * ((1 / distances[current][j]) ** beta) for j in
unvisited)
    for j in unvisited:
        prob = (pheromone[current][j] ** alpha) * ((1 / distances[current][j]) ** beta) / denom
        probabilities.append((j, prob))

    # Roulette wheel selection
    r = random.random()
    cumulative = 0
    for city, prob in probabilities:
        cumulative += prob
        if r <= cumulative:
            return city
    return unvisited[-1]

# -------------------------
# Main ACO Loop
# -------------------------
best_length = float("inf")
best_tour = None

for it in range(iterations):
    all_tours = []
    all_lengths = []

    for ant in range(num_ants):
        start = random.randint(0, num_cities - 1)
        tour = [start]
        unvisited = list(set(range(num_cities)) - {start})

        while unvisited:
            current = tour[-1]
            next_city = select_next_city(current, unvisited)
            tour.append(next_city)
            unvisited.remove(next_city)

        length = tour_length(tour)
        all_tours.append(tour)
        all_lengths.append(length)

        if length < best_length:
            best_length = length
```

```
            best_tour = tour[:]

    # Evaporate pheromones
    for i in range(num_cities):
        for j in range(num_cities):
            pheromone[i][j] *= (1 - rho)

    # Deposit pheromones (each ant contributes)
    for tour, length in zip(all_tours, all_lengths):
        for i in range(len(tour) - 1):
            a, b = tour[i], tour[i + 1]
            pheromone[a][b] += Q / length
            pheromone[b][a] += Q / length
        # close the tour
        pheromone[tour[-1]][tour[0]] += Q / length
        pheromone[tour[0]][tour[-1]] += Q / length

    print(f"Iteration {it+1}: Best Length = {best_length}, Best Tour = {best_tour}")

# -------------------------
# Final Result
# -------------------------
print("\nFinal Best Tour:", best_tour)
print("Final Best Length:", best_length)
```

Output:

```
Iteration 6: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 7: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 8: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 9: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 10: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 11: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 12: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 13: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 14: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 15: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 16: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 17: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 18: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 19: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 20: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 21: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 22: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 23: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 24: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 25: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 26: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 27: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 28: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 29: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 30: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 31: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 32: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 33: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 34: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 35: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 36: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 37: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 38: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 39: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 40: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 41: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 42: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 43: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 44: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 45: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 46: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 47: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 48: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 49: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 50: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]

Final Best Tour: [3, 4, 2, 0, 1]
Final Best Length: 22.35103276995244

=== Code Execution Successful ===
```

Program 5:

Cuckoo Search (CS):

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:

15/07/2025
LAB-05 (a) Cuckoo Search (CS)

Algorithm:
Step 1: Start with a random nest and evaluate their fitness
2. Generate new sol^n for each nest using levy flights
3. Replace a nest if the new sol^n is better
4. Abandon a fraction Pa of worst nests and replace them with new random sol^n
5. Keep the best sol^n found so far and repeat
6. Return the global best solution.

Pseudocode:
input: n = no of nests
Pa = probability of discovery (0.25)
Maxi = Maximum iterations

For t = 1 to maxi do:
  For each nest i:
    - Generate a new solution by levy flight
    - Evaluate fitness of new sol^n

```
for t in range (max_iter):
    new_nests = nests + levy_flight (1.5, (n, dim)
        * (nests - best_nest)
    new_nests = np.clip(new_nests, lb, ub)

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

```
mask = np.random.rand(n) < pa
nests[mask] = np.random.uniform(lb, ub,
    (np.sum(mask), dim))
fitness[mask] = [objective(x) for x in nests
                                    [mask]]
best = nests[np.argmin(fitness)]

def levy_flight(lambda, size):
    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, v = np.random.normal(0, sigma, size),
           np.random.normal(0, 1, size)
    return u / (np.abs(v)**(1/lambda))
```

explore - find new area
exploitation - refine current solutions

Levy flight :- helps exploring the whole search space (global search) instead of getting stuck in one area

input :- n = 25
max_iter = 10
local search - small
global - big
alpha = 0.01 - the jump (step size)
beta = 1.5 - parameter of "levy dist"
Pa = 0.25 - probability of abandoning

Output :-
Best solution : [-0.13401788 -0.00026016]
Best value = 0.01796085 = x^2

Applications:-

1. Image Processing
2. Machine learning and AI
3. Neural Network

CS helps to make predictions as close as possible to the target

- loss function - how wrong the network is?

each nest - set of weights and biases
fitness - loss/error of that candidate set
- abondoning poor nests to generate new sets
- set with lowest loss

weights - how important is the neuron
biases - extra constant value to shift input,
output = (weight * input) + bias

fitness func"

1. Mean Squared Error (MSE)

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

y_i - true target
$\hat{y}_i$ - predicted output
N - no of training samples

Code:

```
import numpy as np
import math

# --------------------------
# Objective Function (minimize)
# --------------------------
def objective_function(x):
    return np.sum(x**2)  # Sphere function example

# --------------------------
# Lévy flight step
# --------------------------
def levy_flight(Lambda, size):
    sigma = (math.gamma(1+Lambda) * math.sin(math.pi*Lambda/2) /
             (math.gamma((1+Lambda)/2) * Lambda * 2**((Lambda-1)/2)))**(1/Lambda)
    u = np.random.normal(0, sigma, size)
    v = np.random.normal(0, 1, size)
    return u / (np.abs(v)**(1/Lambda))

# --------------------------
# Cuckoo Search Algorithm
# --------------------------
def cuckoo_search(n=10, dim=2, lb=-10, ub=10, pa=0.25, max_iter=50):
    # Initialize nests randomly
    nests = np.random.uniform(lb, ub, (n, dim))
    fitness = np.array([objective_function(x) for x in nests])
    best_nest = nests[np.argmin(fitness)].copy()
    best_fitness = np.min(fitness)

    for t in range(max_iter):
        # Generate new solutions via Lévy flights
        new_nests = nests + levy_flight(1.5, (n, dim)) * (nests - best_nest)
        new_nests = np.clip(new_nests, lb, ub)  # keep within bounds
        new_fitness = np.array([objective_function(x) for x in new_nests])

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

        # Abandon some nests (with probability pa)
        abandon = np.random.rand(n) < pa
        nests[abandon] = np.random.uniform(lb, ub, (np.sum(abandon), dim))
        fitness[abandon] = [objective_function(x) for x in nests[abandon]]
```

```
        # Update global best
        if np.min(fitness) < best_fitness:
            best_fitness = np.min(fitness)
            best_nest = nests[np.argmin(fitness)].copy()

        print(f"Iteration {t+1}: Best Fitness = {best_fitness:.6f}")

    return best_nest, best_fitness

# ---------------------------
# Run the algorithm
# ---------------------------
best_solution, best_value = cuckoo_search()
print("\nBest Solution:", best_solution)
print("Best Value:", best_value)
```

Output:

Output

```
Iteration 6: Best Fitness = 3.194164
Iteration 7: Best Fitness = 3.194164
Iteration 8: Best Fitness = 3.194164
Iteration 9: Best Fitness = 1.983263
Iteration 10: Best Fitness = 1.983263
Iteration 11: Best Fitness = 0.816409
Iteration 12: Best Fitness = 0.735204
Iteration 13: Best Fitness = 0.735204
Iteration 14: Best Fitness = 0.735204
Iteration 15: Best Fitness = 0.735204
Iteration 16: Best Fitness = 0.310402
Iteration 17: Best Fitness = 0.310402
Iteration 18: Best Fitness = 0.310402
Iteration 19: Best Fitness = 0.310402
Iteration 20: Best Fitness = 0.299307
Iteration 21: Best Fitness = 0.251654
Iteration 22: Best Fitness = 0.251654
Iteration 23: Best Fitness = 0.206784
Iteration 24: Best Fitness = 0.206784
Iteration 25: Best Fitness = 0.206784
Iteration 26: Best Fitness = 0.206784
Iteration 27: Best Fitness = 0.206784
Iteration 28: Best Fitness = 0.206784
Iteration 29: Best Fitness = 0.206784
Iteration 30: Best Fitness = 0.075533
Iteration 31: Best Fitness = 0.075533
Iteration 32: Best Fitness = 0.075533
Iteration 33: Best Fitness = 0.075533
Iteration 34: Best Fitness = 0.075533
Iteration 35: Best Fitness = 0.075533
Iteration 36: Best Fitness = 0.075533
Iteration 37: Best Fitness = 0.075533
Iteration 38: Best Fitness = 0.075533
Iteration 39: Best Fitness = 0.075533
Iteration 40: Best Fitness = 0.075533
Iteration 41: Best Fitness = 0.075533
Iteration 42: Best Fitness = 0.075533
Iteration 43: Best Fitness = 0.075533
Iteration 44: Best Fitness = 0.075533
Iteration 45: Best Fitness = 0.017961
Iteration 46: Best Fitness = 0.017961
Iteration 47: Best Fitness = 0.017961
Iteration 48: Best Fitness = 0.017961
Iteration 49: Best Fitness = 0.017961
Iteration 50: Best Fitness = 0.017961

Best Solution: [-0.13401788 -0.00026016]
Best Value: 0.017960859555768195

--- Code Execution Successful ---
```

Program 6:

Grey Wolf Optimizer (GWO):

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 c

Algorithm:

29/9/2025

LAB-06 (5) Grey Wolf optimizer

Algorithm:-
- best wolf - alpha (α) leader, best sol^n found so far
- the second best - beta (β)
- the third best - delta (δ)
- the rest - omega (ω) followers

Step1:- Start with random wolves (random guesses)
Step2:- Evaluate how good each wolf is (fitness)
Step3:- Alpha, Beta, Delta guide the rest. Each wolf updates its position based on these 3 leaders.
Step4:- Repeat until wolves close in on prey (best sol^n)
Step5:- Finally, return alpha as best solution.

Pseudo Code:-

1. Initialize population of wolves.
2. Evaluate fitness of each wolf.
3. Identify alpha, beta, delta.
4. For iteration = 1 to maxi
   a = 2 - (2 * iteration / maxi)
   For each wolf:
   For each leader (α, β, δ)
   Generate random nos r1, r2
   A = 2 * a * r1 - a
   C = 2 * r2
   D = |C * leader_pos^n - wolf pos^n)
   X_candidate = leader_pos^n - A * D
   Update wolf posit^n = (X_α + X_β + X_δ)/3
   Check boundaries

Evaluate new fitness of wolves
Update α, β, δ
Output best sol^n (α) and its fitness

Output: It 1 : Best Fitness . 1.659563
2 ——— 0.4865641
3
⋮
50 : ——— 0.00
Best Position - [-8.53962971e-16 -8.09849340e-16]
Best Fitness : 2.21293301 95 e-31

Applications:-
1. Machine learning
2. Neural Networks
3. Power Systems and smart Grids
3 Generators - $G_1 - C_1(P_1) = 0.1P_1^2 + 5P_1 + 100$
$G_2 - C_2(P_2) = 0.08P_2^2 + 4P_2 + 120$
$G_3 - C_3(P_3) = 0.12P_3^2 + 6P_3 + 90$
constraint - $P_1 + P_2 + P_3 = P_{demand}$
Objective fun^n - $f(P_1, P_2, P_3) = C_1(P_1) + C_2(P_2) + C_3(P_3)$
aim - find best way to run the system efficiently and cheaply
wolf $[P_1, P_2, P_3]$
update posi^n - $\vec{X}(t+1) = (\vec{X_1} + \vec{X_2} + \vec{X_3})/3$
→ decrease overtime
A - coefficient vector controlling step size toward / away
dont get stuck in local opt^n ← C - ——— adding randomness to leaders influence
D - dist^n b/w leader and wolf, scaled by C

MG
29/9/25

Code:

```
import numpy as np

# Objective Function (Sphere function)
def objective_function(x):
    return np.sum(x**2)

# Grey Wolf Optimizer
def grey_wolf_optimizer(obj_func, dim=2, search_agents=10, max_iter=50, lb=-10, ub=10):
    # Initialize wolf positions randomly
    wolves = np.random.uniform(lb, ub, (search_agents, dim))
    fitness = np.array([obj_func(w) for w in wolves])

    # Identify alpha, beta, delta wolves
    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    # Find initial alpha, beta, delta
    for i in range(search_agents):
        if fitness[i] < alpha_score:
            alpha_score = fitness[i]
            alpha = wolves[i].copy()
        elif fitness[i] < beta_score:
            beta_score = fitness[i]
            beta = wolves[i].copy()
        elif fitness[i] < delta_score:
            delta_score = fitness[i]
            delta = wolves[i].copy()

    # Main loop
    for t in range(max_iter):
        a = 2 - t * (2 / max_iter)  # linearly decreases from 2 to 0

        for i in range(search_agents):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()

                A1, C1 = 2 * a * r1 - a, 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, C2 = 2 * a * r1 - a, 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, C3 = 2 * a * r1 - a, 2 * r2
```

```
            D_delta = abs(C3 * delta[j] - wolves[i][j])
            X3 = delta[j] - A3 * D_delta

            wolves[i][j] = (X1 + X2 + X3) / 3  # update wolf position

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

        # Fitness evaluation
        score = obj_func(wolves[i])

        if score < alpha_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = alpha_score, alpha.copy()
            alpha_score, alpha = score, wolves[i].copy()
        elif score < beta_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = score, wolves[i].copy()
        elif score < delta_score:
            delta_score, delta = score, wolves[i].copy()

    print(f"Iteration {t+1}: Best Fitness = {alpha_score:.6f}")

  return alpha, alpha_score

# Run GWO
best_position, best_value = grey_wolf_optimizer(objective_function, dim=2, search_agents=15, max_iter=50)
print("\nBest Position:", best_position)
print("Best Fitness:", best_value)
```

Output:

Output

```
Iteration 6: Best Fitness = 0.001403
Iteration 7: Best Fitness = 0.001236
Iteration 8: Best Fitness = 0.000012
Iteration 9: Best Fitness = 0.000012
Iteration 10: Best Fitness = 0.000003
Iteration 11: Best Fitness = 0.000000
Iteration 12: Best Fitness = 0.000000
Iteration 13: Best Fitness = 0.000000
Iteration 14: Best Fitness = 0.000000
Iteration 15: Best Fitness = 0.000000
Iteration 16: Best Fitness = 0.000000
Iteration 17: Best Fitness = 0.000000
Iteration 18: Best Fitness = 0.000000
Iteration 19: Best Fitness = 0.000000
Iteration 20: Best Fitness = 0.000000
Iteration 21: Best Fitness = 0.000000
Iteration 22: Best Fitness = 0.000000
Iteration 23: Best Fitness = 0.000000
Iteration 24: Best Fitness = 0.000000
Iteration 25: Best Fitness = 0.000000
Iteration 26: Best Fitness = 0.000000
Iteration 27: Best Fitness = 0.000000
Iteration 28: Best Fitness = 0.000000
Iteration 29: Best Fitness = 0.000000
Iteration 30: Best Fitness = 0.000000
Iteration 31: Best Fitness = 0.000000
Iteration 32: Best Fitness = 0.000000
Iteration 33: Best Fitness = 0.000000
Iteration 34: Best Fitness = 0.000000
Iteration 35: Best Fitness = 0.000000
Iteration 36: Best Fitness = 0.000000
Iteration 37: Best Fitness = 0.000000
Iteration 38: Best Fitness = 0.000000
Iteration 39: Best Fitness = 0.000000
Iteration 40: Best Fitness = 0.000000
Iteration 41: Best Fitness = 0.000000
Iteration 42: Best Fitness = 0.000000
Iteration 43: Best Fitness = 0.000000
Iteration 44: Best Fitness = 0.000000
Iteration 45: Best Fitness = 0.000000
Iteration 46: Best Fitness = 0.000000
Iteration 47: Best Fitness = 0.000000
Iteration 48: Best Fitness = 0.000000
Iteration 49: Best Fitness = 0.000000
Iteration 50: Best Fitness = 0.000000

Best Position: [-3.53962974e-16 -3.09844340e-16]
Best Fitness: 2.2129330195336707e-31
```

Problem 7:

Parallel Cellular Algorithms and Programs:

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 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.

Algorithm:

13/10/25
LAB - 07 ⑥ Parallel Cellular Algorithms & Programs

Algorithm:-

Step1:- Initialize Parameters :-
- Grid size → total cells = potential sol^n.
- Neighborhood structure (Von Neumann up, down, left, right / Moore 8 neighbors)
- Max iterations

Step2:- Initialize population
- Assign each cell a random sol^n in search space

Step3:- Evaluate fitness
- Compare fitness for each cell using the func^n

Step4:- update states (parallel update)
- For each cell
  compare its sol^n with neighbor's sol^n
  move towards best sol^n
  apply randomness to keep diversity

Step5:- Repeat the steps. After all it best cell = final sol^n

Pseudo Code:-

```
def fitness (x):
    return x**2
for each cell
    fitness[cell] = f(solution[cell])

for it in range (maxi):
    for each cell in grid (parallel update):
        neighbors = get-neighbors(cell)
        best neighbor = argmin (fitness [neighbors])
        sol[cell] = move-towards (sol[cell],
                                  sol[best-neighbor])
```

sol[cell] = mutate(sol[cell])
fitness[cell] = f(sol[cell])
update global_sol^n
return global_sol^n, global_fitness

Main point :- Divide the problems into many small interacting parts, process them in parallel, let local interactions lead to a global sol^n

Output: rows, cols = 8, 8 - grid size
dim = 2 - dimensionality of problem
maxi = 50 no of iterations
move_strength = 0.2 - step size toward best neighbor
mutation_sigma = 0.05 - random mutation strength
function = x ** 2
It 1 : Best Fitness: 0.131066, Position = [0.365, -0.194]
:
It 50: ——————— = 0.000442, Pos: [-0.019, -0.007]
Best sol^n found:
Position: [-0.0196 -0.0075]
Fitness: 0.000044

Applications :-
1. Disease spread & epidemic modelling
2. Music & Art Generation
3. Machine learning & Image Processing.
4. Cryptography & Security
* PCA used to generate strong pseudo-random sequence
+ used for secure key generation - encryption and intrusion detection in networks.

- seed initialization: random binary vector
cells - bits in the seed
parallel update - generate pseudo random key stream
output - encrypt key

Output :- Plaintext : HELLO
Plaintext bits [01001000100010101001100010011000 1001111]
initial seed [00010000]
Generated key bits [00001000000110000011011 0000010
ciphertext bits: [01 001110010000110100101001 0010 01010010010011]
decrypted bits: [01001000 100001010100110001001 10001001111]
decrypted text : HELLO

- each char to 8 bits using ASCII
- Generated key bits: using left shift, right shift and operator using its neighbor states.
- ciphertext xor plaintext & key bits
- decrypt based on ASCII
- decrypt list xor ciphertext & keybits

MG
13/10/25

Code:

```
import numpy as np

# -------------------------
# Cellular Automata Parameters
# -------------------------
n = 8  # number of cells in CA
rule_number = 30  # Wolfram rule 30

# -------------------------
# Apply CA rule
# -------------------------
def apply_rule(left, center, right, rule):
    index = (left << 2) | (center << 1) | right
    return (rule >> index) & 1

# -------------------------
# Generate CA key stream
# -------------------------
def generate_key(seed, rule, length):
    state = seed.copy()
    key_stream = []
    for _ in range(length):
        key_stream.append(state[-1])  # output last cell
        new_state = np.zeros_like(state)
        for i in range(len(state)):
            left = state[i-1] if i>0 else state[-1]
            center = state[i]
            right = state[i+1] if i<len(state)-1 else state[0]
            new_state[i] = apply_rule(left, center, right, rule)
        state = new_state
    return np.array(key_stream)

# -------------------------
# XOR for encryption/decryption
# -------------------------
def xor_bits(data_bits, key_bits):
    return np.array([d ^ k for d, k in zip(data_bits, key_bits)])

# -------------------------
# Convert string to bits and back
# -------------------------
def string_to_bits(s):
    bits = []
    for char in s:
        bits.extend([int(b) for b in format(ord(char), '08b')])
    return bits
```

```
def bits_to_string(bits):
    chars = []
    for i in range(0, len(bits), 8):
        byte = bits[i:i+8]
        chars.append(chr(int(''.join(map(str, byte)), 2)))
    return ''.join(chars)

# -------------------------
# Main
# -------------------------
plaintext_str = "HELLO"
plaintext_bits = string_to_bits(plaintext_str)
print("Plaintext:", plaintext_str)
print("Plaintext bits:", plaintext_bits)

# Random CA seed of 8 bits
seed = np.random.randint(0, 2, n)
print("Initial CA Seed:   ", seed.tolist())

# Repeat key stream to match plaintext length
key_stream = np.tile(generate_key(seed, rule_number, n), len(plaintext_bits)//n + 1)[:len(plaintext_bits)]
print("Generated Key Bits:", key_stream.tolist())

# Encrypt
ciphertext_bits = xor_bits(plaintext_bits, key_stream)
print("Ciphertext bits:   ", ciphertext_bits.tolist())

# Decrypt
decrypted_bits = xor_bits(ciphertext_bits, key_stream)
decrypted_str = bits_to_string(decrypted_bits)
print("Decrypted bits:    ", decrypted_bits.tolist())
print("Decrypted Text:    ", decrypted_str)
```

Output:

main.py

```
46      return bits
47
48  def bits_to_string(bits):
49      chars = []
50      for i in range(0, len(bits), 8):
51          byte = bits[i:i+8]
52          chars.append(chr(int(''.join(map(str, byte)), 2)))
53      return ''.join(chars)
54
55  # -------------------------
56  # Main
57  # -------------------------
58  plaintext_str = "HELLO"
59  plaintext_bits = string_to_bits(plaintext_str)
60  print("Plaintext:", plaintext_str)
61  print("Plaintext bits:", plaintext_bits)
62
63  # Random CA seed of 8 bits
64  seed = np.random.randint(0, 2, n)
65  print("Initial CA Seed:   ", seed.tolist())
66
67  # Repeat key stream to match plaintext length
68  key_stream = np.tile(generate_key(seed, rule_number, n), len(plaintext_bits)//n
        (plaintext_bits)]
69  print("Generated Key Bits:", key_stream.tolist())
70
71  # Encrypt
72  ciphertext_bits = xor_bits(plaintext_bits, key_stream)
73  print("Ciphertext bits:   ", ciphertext_bits.tolist())
74
75  # Decrypt
76  decrypted_bits = xor_bits(ciphertext_bits, key_stream)
77  decrypted_str = bits_to_string(decrypted_bits)
78  print("Decrypted bits:    ", decrypted_bits.tolist())
79  print("Decrypted Text:    ", decrypted_str)
80
```