VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sushma B T(1WA23CS002)

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

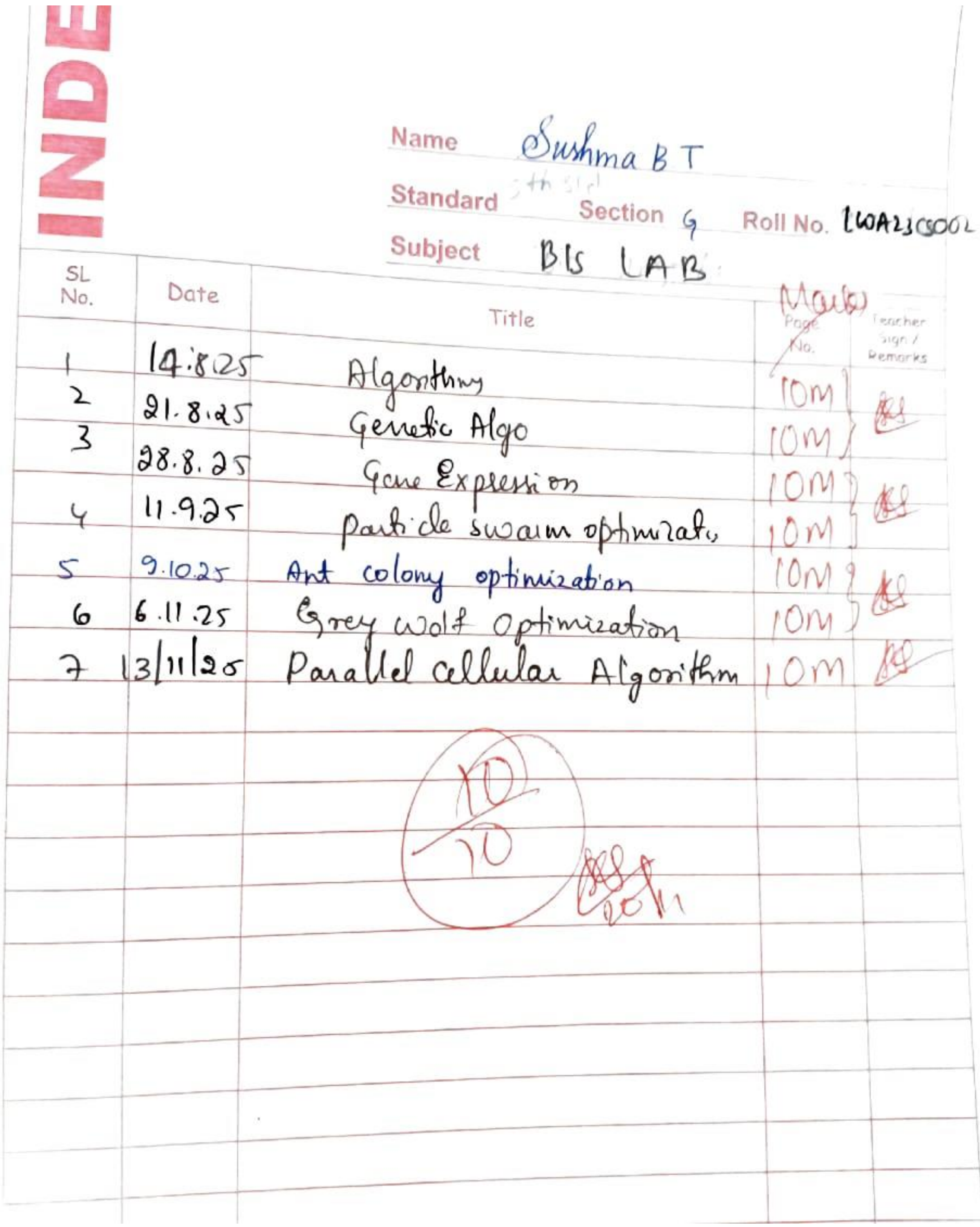
CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Sushma B T (1WA23CS002),** 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.

Dr.Shwetha KS Assistant Professor Department of CSE, BMSC	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

6	6/11/25	Grey Wolf Optimizer (GWO)	38-44
7	13/11/25	Parallel Cellular Algorithms and Programs	45-49



INDEX

Name Sushma B T

Standard 5th sem Section G Roll No. 1WA23CS002

Subject BIS LAB

SL No.	Date	Title	Marks	Teacher Sign / Remarks
1	14.8.25	Algorithms	10M	
2	21.8.25	Genetic Algo	10M	
3	28.8.25	Gene Expression	10M	
4	11.9.25	Particle swarm optimizati	10M	
5	9.10.25	Ant colony optimization	10M	
6	6.11.25	Grey wolf optimization	10M	
7	13/11/25	Parallel cellular Algorithm	10M	

10/10

Github Link:
https://github.com/Sushmabt680/BIS

Program 1 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.

2. Genetic Algorithm (GA)

→ Is an optimization and search technique based on the principles of natural selection and genetics

→ It belongs to the class of evolutionary algm.

Genetic algorithm

→ Select Initial population.
→ Calculate the fitness
→ Selecting mating pool
→ Crossover
→ Mutation
→ Replacement → Termination.

Initial population is consider for the given value of x -[0-31]

String no	Initial population	x value	fitness func $f(x)=x^2$	Prob
1	01100	12	144	0.124
2	11001	25	625	0.541
3	00101	5	25	0.0216

Crossover

→ Single - point crossover
→ Multi point crossover
→ Uniform crossover

SPC:-

Parents

P1 = 11100 (28)
P2 = 10101 (21)

Choose crossover point: after 2nd bit

P1 = 11|100
P2 = 10|101

• Children:

C1 = 11 + 101 = 11101 (29)
C2 = 10 + 100 = 10100 (20)
} offspring

Two point crossover after 1st and 4th bit

P1 = 01010 (10)
P2 = 11001 (25)

P1' = 0|101|0
P2 1|100|1

% Prob Expected count

12.47 0.4987→1 Swap the middle part

54.11 2.1645→2 C1 = 0 + 100 + 0 = 01000 (8)

2.16 0.0866→0 C2 = 1 + 101 + 1 = 11011 (27)

31.26 1.25→1 New offspring 8 and 27)

Uniform crossover

bits → 50% ~~from~~ from P1
50% from P2

$$\text{Expected} = \frac{f(x_i)}{avg(\Sigma(fx))}$$

3) Selecting mating pool

String no	mating pool	crossover point	offspring after crossover	x value	fitness
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289

sum 1763

avg 440.75

max 729

5) Mutation

String no	offspring after crossover	Mutation chromosome for flipping	offspring after mutation	x value	fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00100	10100	20	400

Code:

```
import random

CHROM_LENGTH = 5
CROSS_RATE = 0.8
MUT_RATE = 0.1

def fitness(x):
    return x**2

def encode(x):
    return format(x, f'0{CHROM_LENGTH}b')

def decode(b):
    return int(b, 2)

def roulette_selection(pop, fitnesses):
    total_fit = sum(fitnesses)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return pop[i]
    return pop[-1]

def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, CHROM_LENGTH-1)
        c1 = p1[:point] + p2[point:]
        c2 = p2[:point] + p1[point:]
        return c1, c2
    return p1, p2

def mutate(chrom):
    chrom_list = list(chrom)
    for i in range(CHROM_LENGTH):
        if random.random() < MUT_RATE:
            chrom_list[i] = '1' if chrom_list[i] == '0' else '0'
    return ''.join(chrom_list)

def genetic_algorithm():
    user_input = input("Enter initial population values (space-separated, e.g. 12 23 5 19): ")
    values = list(map(int, user_input.split()))
```

```python
    generations = int(input("Enter number of generations to run: "))

    POP_SIZE = len(values)
    population = [encode(x) for x in values]

    print("\nInitial Population:", population, [decode(c) for c in population])

    global_best = population[0]
    global_best_fit = fitness(decode(global_best))

    for gen in range(1, generations + 1):
        decoded = [decode(c) for c in population]
        fitnesses = [fitness(x) for x in decoded]

        best_idx = fitnesses.index(max(fitnesses))
        if fitnesses[best_idx] > global_best_fit:
            global_best = population[best_idx]
            global_best_fit = fitnesses[best_idx]

        total_fit = sum(fitnesses)
        probs = [f / total_fit for f in fitnesses]
        expected = [p * POP_SIZE for p in probs]

        print(f"\nGeneration {gen}")
        for i in range(POP_SIZE):
            print(f"x={decoded[i]}, bin={population[i]}, fit={fitnesses[i]}, "
                  f"prob={probs[i]:.3f}, exp_count={expected[i]:.2f}")

        new_pop = [global_best]

        while len(new_pop) < POP_SIZE:
            p1 = roulette_selection(population, fitnesses)
            p2 = roulette_selection(population, fitnesses)
            c1, c2 = crossover(p1, p2)
            c1, c2 = mutate(c1), mutate(c2)
            new_pop.extend([c1, c2])

        population = new_pop[:POP_SIZE]

    decoded = [decode(c) for c in population]
    fitnesses = [fitness(x) for x in decoded]
    best_idx = fitnesses.index(max(fitnesses))
    print("\nFinal Best Solution:", decoded[best_idx], population[best_idx], "fitness=",
fitnesses[best_idx])
```

genetic_algorithm()
OUTPUT:

```
=== RESTART: C:/Users/student/AppData/Local/Programs/Python/Python313/gene.py ==
Initial Population: ['01100', '11001', '00101', '10011'] [12, 25, 5, 19]

Generation 1
x=12, bin=01100, fit=144, prob=0.125, exp_count=0.50
x=25, bin=11001, fit=625, prob=0.541, exp_count=2.16
x=5, bin=00101, fit=25, prob=0.022, exp_count=0.09
x=19, bin=10011, fit=361, prob=0.313, exp_count=1.25

Generation 2
x=19, bin=10011, fit=361, prob=0.211, exp_count=0.85
x=19, bin=10011, fit=361, prob=0.211, exp_count=0.85
x=25, bin=11001, fit=625, prob=0.366, exp_count=1.46
x=19, bin=10011, fit=361, prob=0.211, exp_count=0.85

Generation 3
x=17, bin=10001, fit=289, prob=0.137, exp_count=0.55
x=19, bin=10011, fit=361, prob=0.171, exp_count=0.68
x=25, bin=11001, fit=625, prob=0.295, exp_count=1.18
x=29, bin=11101, fit=841, prob=0.397, exp_count=1.59

Generation 4
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20
x=25, bin=11001, fit=625, prob=0.384, exp_count=1.54
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20
x=29, bin=11101, fit=841, prob=0.517, exp_count=2.07

Generation 5
x=29, bin=11101, fit=841, prob=0.517, exp_count=2.07
x=25, bin=11001, fit=625, prob=0.384, exp_count=1.54
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20

Final Best Solution: 29 11101 fitness= 841
```

Algorithm:

```
import random
CHROM_LENGTH = 5
POP_SIZE = 4
CROSS_RATE = 0.8
MUT_RATE = 0.1

def fitness(x):
    return x**2

def encode(x):
    return format(x, f'0{CHROM_LENGTH}b')
def decode(b):
    return int(b, 2)
def roulette_selection(pop, fitnesses):
    total_fit = sum(fitnesses)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return pop[i]
    return pop[-1]

def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, CHROM_LENGTH-1)
        c1 = p1[:point] + p2[point:]
        c2 = p2[:point] + p1[point:]
        return c1, c2
    return p1, p2
```

```
def mutate (chrom):
    chrom_list = list(chrom)
    for i in range (CHROM_LENGTH):
        if random.random() < MUT_RATE:
            chrom_list [i] = '1'
                if chrom_list[i]=='0' else '0'
            return ''.join(chrom_list)
def genetic_algorithm():
    population = [encode(x) for x in[12, 23, 5, 19]]
        print ("Initial Population:", population,[decode(c
          for c in population])

for gen in range (1, 6):
    decode = [decode(c) for c in population]
    fitnesses = [fitness(x) for x in decoded]
    total_fit = sum (fitnesses)
    probs = [f / total_fit for f in fitnesses]
    expected = [p * POP_SIZE for p in probs]
    print (f"\n Generation {gen}")
      for i in range (POP_SIZE):
        print (f"x = {decoded[i]},
ben = { population [i]}, fit = {fitnesses[i]}")
        f" prob = {probs [i]:.3f}, exp_count =
        {expected[i]:.2f}")
        new_pop = [ ]
        while len(new_pop) < POP_SIZE:
  p1 = roulette_selection (population, fitnesses)
  p2 = roulette_selection (population, fitnesses)
    c1, c2 = crossover (p1, p2)
    c1, c2 = mutate(c1), mutate(c2)
      new_pop.extend ([c1, c2])
        population = new_pop[: POP_SIZE]
```

decoded = [decode (c) for c in population]
fitnesses = [fitness (x) for x in decoded]
best_idx = fitnesses.index (max (fitnesses))
print ("\n Final Best Solution:", decoded [best_idx], population [best_idx], "fitness =", fitness [best idx])

genetic_algorithm()

Output
Initial Population: ['01100', '10111', '00101', '10011']
[12, 25, 5, 19]
Generation 1
x = 12, bin = 01100, fit = 144, prob = 0.184 exp_count = 0.58
x = 25, bin = 10111, fit = 625 prob = 0.544 exp count = 2.46
x = 5, bin = 00101, fit = 25 prob = 0.029 exp count = 0.09
x = 19, bin = 10011, fit = 361 prob = 0.343 exp count = 1.25

Generation 2
x = 19 bin = 10011 fit = 361 prob = 0.211 exp_count = 0.85
x = 19 bin = 10011 fit = 361 prob = 0.211 exp_count = 0.85
x = 25 bin = 11001 fit = 625 prob = 0.366 exp_count = 1.46
x = 19 bin = 10011 fit = 361 prob = 0.211 exp_count = 0.85

Generation 3
x = 17, bin = 10001, fit = 289, prob = 0.137, exp_count = 0.55
x = 19, bin = 10011, fit = 361, prob = 0.171 exp_count = 0.68
x = 25, bin = 11001, fit = 625, prob = 0.295 exp_count = 1.18
x = 29, bin = 11101, fit = 841, prob = 0.397 exp_count = 1.59

Generation 4:

x = 9, bin = 01001, fit = 0.050, exp_count = 0.20
x = 25, bin = 11001, fit = 0.384, exp_count = 1.54
x = 9, bin = 01001, fit = 0.050, exp_count = 0.20
x = 29, bin = 11101, fit = 0.517, exp_count = 2.07

Generation 5
x = 29, bin = 11101, fit 841, prob = 0.517, exp_count = 2.07
x = 25, bin = 11001, fit = 625 prob = 0.384 exp_count = 1.54
x = 9, bin = 01001 fit = 81, prob = 0.050 exp count = 0.20
x = 9, bin = 01001 fit = 81, prob = 0.050 exp count = 0.20

Final Best Solution: 29 11101 fitness = 841

Problem 2
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

```
import random

POP_SIZE = 4
CHROM_LENGTH = 5
MAX_GENERATIONS = 5
MUTATION_RATE = 0.1

def gene_expression(chromosome):
    return int(chromosome, 2)

def fitness(chromosome):
    x = gene_expression(chromosome)
    return x * 2* x

def get_population_from_input():
    population = []
    print(f"Enter {POP_SIZE} chromosomes (each {CHROM_LENGTH} bits, only 0 or 1):")
    while len(population) < POP_SIZE:
        chrom = input(f"Chromosome {len(population) + 1}: ").strip()
        if len(chrom) == CHROM_LENGTH and all(c in '01' for c in chrom):
            population.append(chrom)
        else:
            print(f"Invalid chromosome! Please enter exactly {CHROM_LENGTH} bits (0 or 1).")
    return population

def select(population):
    fitnesses = [fitness(chrom) for chrom in population]
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for i, chrom in enumerate(population):
        current += fitnesses[i]
        if current > pick:
            return chrom
```

```
def crossover(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):
    mutated = ''
    for bit in chromosome:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

def genetic_algorithm():
    population = get_population_from_input()
    print(f"Initial Population: {population}")

    best_overall = None
    best_fitness = float('-inf')

    for generation in range(MAX_GENERATIONS):
        new_population = []
        while len(new_population) < POP_SIZE:
            parent1 = select(population)
            parent2 = select(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POP_SIZE]

        best = max(population, key=fitness)
        best_fit = fitness(best)
        if best_fit > best_fitness:
            best_fitness = best_fit
            best_overall = best

        print(f"Generation {generation + 1}: Best Chromosome = {best}, "
              f"Expressed Value = {gene_expression(best)}, Fitness = {best_fit}")

    print(f"\nBest solution after {MAX_GENERATIONS} generations: {best_overall} "
```

```
        f"with expressed value = {gene_expression(best_overall)} and fitness = {best_fitness}")

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

OUTPUT:

```
Enter 4 chromosomes (each 5 bits, only 0 or 1):
Chromosome 1: 11001
Chromosome 2: 01101
Chromosome 3: 00110
Chromosome 4: 10011
Initial Population: ['11001', '01101', '00110', '10011']
Generation 1: Best Chromosome = 11101, Expressed Value = 29, Fitness = 1682
Generation 2: Best Chromosome = 11011, Expressed Value = 27, Fitness = 1458
Generation 3: Best Chromosome = 11001, Expressed Value = 25, Fitness = 1250
Generation 4: Best Chromosome = 11010, Expressed Value = 26, Fitness = 1352
Generation 5: Best Chromosome = 11011, Expressed Value = 27, Fitness = 1458

Best solution after 5 generations: 11101 with expressed value = 29 and fitness = 1682
```

Lab 3

Optimization via Gene Expression

```
import random

CHROM LENGTH = 5
POP_SIZE = 4
CROSS_RATE = 0.8
MUT_RATE = 0.1
GENERATIONS = 5

def fitness(x):
    return x **2
def encode(x):
    return format(x, f'0{CHROM_LENGTH}b')
def decode(b):
    return int(b,2)
def roulette_selection(pop, fitnesses):
    total_fit = sum(fitnesses)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return pop[i]
    return pop[-1]
def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, CHROM_LENGTH-1)
        c1 = p1[:point] + p2[point:]
        c2 = p2[:point] + p1[point:]
        return c1, c2
    return p1, p2
```

```
def mutate (chrom):
  chrom_list = list(chrom)
  for i in range (POP_SICHROM LENGTH):
      if random.random() < MUT_RATE:
          chrom_list[i] = '1' if chrom_list[i] == '0' else
      return ''.join(chrom_list)
def initialize_population():
  population = [ ]
  for _ in range (POP_SIZE):
        rand_int = random.randint(0, 2**CHROM_LENGTH)
        population.append(encode(rand_int))
  return population
def gene_expression_algorithm():
 population = initialize_population()
 print("Initial population:", population, [decode(c) for
          c in population])
 for gen in range(1, generations+1):
    decoded = [decode(c) for c in population]
    fitnesses = [fitness(x) for x in decoded]

    total_fit = sum(fitnesses)
    probs = [f / total_fit for f in fitnesses]
    expected = [p * POP_SIZE for p in probs]

    print(f"\n Generation {gen}")
    for i in range(POP_SIZE):
       print(f"x = {decoded[i]}, bin = {population[i]},
        fit = {fitnesses[i]}, " f" prob = {probs[i]:.3f},
           exp_count = {expected[i]:.2f}")
new_pop = []
while len(new_pop) < POP_SIZE:
 p1 = roulette_selection(population, fitnesses)
```

p2 = roulette_selection(population, fitnesses)
c1, c2 = crossover(p1, p2)
c1, c2 = mutate(c1), mutate(c2)
new_pop.extend([[c1, c2]])

population - new-pop[: POP_SIZE]
decoded =[decode(c) for c in population]
fitnesse = [fitnesses.index(max(fitnesses))
print("\n Final Best solution:", decoded [best_idx],
population best_idx], " fitness=", fitnesses [best_idx])
genetic_algorithm()

~~Generation 1~~
~~x = 45~~
x = 8
x = 4
x = 28

~~Generation~~
Output:
Enter 4 chromosomes (each 5 bits, only 0 or 1):
chromosome1: 01100
chromosome2: 10111
Chromosome3: 00101
chromosome 4: 10011
Initial population: ['01100', '10111', '00101', '10011']
Generation1: Best Chromosome= 10111, Expressed Value=23
Fit = 1058
Generation2: Best Chromosome= 111171, Expressed Value = 31,
Fit = 1922
Generation3: Best Chromosome = 11111, Expressed Value = 31,
Fit 1922.

Generation 4:
Best Chromosome = 10111, Expressed Value = 23 Fitness = 1057
Generation 5
Best chromosome = 11011, Expressed Value = 27, Fitness = 1457

Best solution after 5 generation 11111 with expressed value = 31 and fitness = 1922

Problem 3
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.

```
import numpy as np

num_drones = 5
area_size = 20
num_particles = 30
iterations = 10

w = 0.5
c1 = 1.5
c2 = 1.5

dim = num_drones * 2

particles = np.random.uniform(0, area_size, (num_particles, dim))
velocities = np.zeros((num_particles, dim))

pbest_positions = particles.copy()
pbest_scores = np.full(num_particles, -np.inf)

gbest_position = None
gbest_score = -np.inf
```

```
def fitness(position):
    x = np.sum(position)
    return (x*2 + 5*x + 20)

for iter in range(iterations):
    for i in range(num_particles):
        score = fitness(particles[i])

        if score > pbest_scores[i]:
            pbest_scores[i] = score
            pbest_positions[i] = particles[i].copy()

        if score > gbest_score:
            gbest_score = score
            gbest_position = particles[i].copy()

    r1, r2 = np.random.rand(), np.random.rand()
    for i in range(num_particles):
        velocities[i] = (w * velocities[i] +
                         c1 * r1 * (pbest_positions[i] - particles[i]) +
                         c2 * r2 * (gbest_position - particles[i]))

        particles[i] += velocities[i]
        particles[i] = np.clip(particles[i], 0, area_size-1)

    print(f"Iteration {iter}, Best Fitness: {gbest_score:.2f}")

print("\nOptimized drone waypoints (x,y):")
optimized_coords = gbest_position.reshape((num_drones, 2))
for idx, coord in enumerate(optimized_coords):
    print(f"Drone {idx+1}: {coord}")
```
OUTPUT:

```
Iteration 0, Best Fitness: 995.60
Iteration 1, Best Fitness: 995.60
Iteration 2, Best Fitness: 995.60
Iteration 3, Best Fitness: 1062.14
Iteration 4, Best Fitness: 1109.77
Iteration 5, Best Fitness: 1126.03
Iteration 6, Best Fitness: 1131.01
Iteration 7, Best Fitness: 1143.85
Iteration 8, Best Fitness: 1151.87
Iteration 9, Best Fitness: 1155.98

Optimized drone waypoints (x,y):
Drone 1: [19. 19.]
Drone 2: [13.47083229 13.14074042]
Drone 3: [ 9.53393196 19.        ]
Drone 4: [19. 19.]
Drone 5: [19.         12.13802119]
```

Lab 4

Particle Swarm optimization

1) Objective function
2) P best → all pos
3) g best → out g all
4) v_i^{t+1}
5) x_i^{t+1}

Power system optimization

```
import random
import numpy as np
cost_coeffs = np.array([
        [0.004, 5.3, 500],
        [0.006, 5.5, 400],
        [0.009, 5.8, 200]
  ])
 gen_limits = np.array([
        [100,600], [100,400], [50,200]])
        [
  p_demand = 850

# PSO parameters
num_particles = 30
max_iter = 5
w = 0.7
c1 = 1.5
c2 = 1.5
positions = np.random.uniform(gen_limits[:,0], gen_limits[:,1],
                (num_particles, len(gen_limits)))
 velocities = np.zeros_like(positions)
```

```
def cost - function(p):
    return np.sum(cost-coeffs[:,0] * p ** 2 + cost-coeffs[:,1]
      * p + cost-coeffs[:2], axis = 1

pbest-positions = positions.copy()
pbest-scores = np.array([total-cost(p) for p in positions])

gbest-index = np.argmin(pbest-scores)
gbest-position = pbest-positions[gbest-index].copy()
gbest-score = pbest-scores[gbest-index]

for iteration in range(max-iter):
   r1 = np.random.rand(num-particles, len(gen-limits))
   r2 = np.random.rand(numparticles, len(gen-limits))

velocities = (w * velocities + c1 * r1 * (personal-best-position-
              positions) + c2 * r2 * (global-best-position-positions)

positions + = velocities

cost = cost-functions(positions)
  for iteration in range(num-iterations):
     for i in range(num-particles):
        fitness = total-cost(positions[i]

  if fitness < pbest-scores[i]:
     pbest scores[i] = fitness
     pbest-position[i] = positions[i].copy()

min-idx = np.argmin(pbest-scores)
if pbest-scores[min idx] < gbest-score:
   gbest-score = pbest scores[min-idx]
   gbest-position = pbest-positions[min-idx-copy()
```

Bafna Gold

```
print(f"Iteration {iteration+1}: Best Cost = {global_best
        cost:.2f}")
    print("\n Optimal Generation after 5 iterations:")
      for i, power in enumerate(global_best_position):
        print(f"Generator {i+1}: {power:.2f} MW")
    print(f"Total Cost: {global_best_cost:.2f}")
```

Output

```
Iteration 1: Best cost = 6321.22
Iteration 2:           = 6221.15
Iteration 3:           = 6221.15
Iteration 4:           = 6221.15
Iteration 5:           = 6221.15

Optimal Generation after 5 Iterations:
  Generator 1: 115.16 MW
  Generator 2: 378.15 MW
  Generator 3: 200.00 MW
Total cost: 6221.15
```

Problem 4

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

```
import numpy as np
import random
```

```python
NUM_ANTS =2
NUM_ITERATIONS = 50
ALPHA = 1.0
BETA = 5.0
EVAPORATION = 0.5
Q = 100

def input_matrix(name):
    print(f"Enter the {name} matrix row by row (space-separated). Type 'done' when finished:")
    matrix = []
    while True:
        row = input()
        if row.strip().lower() == 'done':
            break
        row_values = list(map(float, row.strip().split()))
        matrix.append(row_values)
    return np.array(matrix)

print("Input Cost Matrix (Distance Matrix):")
dist_matrix = input_matrix("cost")
NUM_CITIES = len(dist_matrix)

print("\nInput Initial Pheromone Matrix:")
pheromone = input_matrix("pheromone")

assert dist_matrix.shape == (NUM_CITIES, NUM_CITIES), "Cost matrix must be square."
assert pheromone.shape == (NUM_CITIES, NUM_CITIES), "Pheromone matrix must be
square."

best_distance = float('inf')
best_path = []

for iteration in range(NUM_ITERATIONS):
    all_paths = []
    all_distances = []

    for ant in range(NUM_ANTS):
        path = [random.randint(0, NUM_CITIES - 1)]

        while len(path) < NUM_CITIES:
            current_city = path[-1]
            probabilities = []

            for next_city in range(NUM_CITIES):
```

```
            if next_city not in path:
                tau = pheromone[current_city][next_city] ** ALPHA
                eta = (1 / dist_matrix[current_city][next_city]) ** BETA
                probabilities.append(tau * eta)
            else:
                probabilities.append(0)

        probabilities = np.array(probabilities)
        probabilities_sum = probabilities.sum()
        if probabilities_sum == 0:
            break
        probabilities /= probabilities_sum

        next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
        path.append(next_city)

    if len(path) < NUM_CITIES:
        continue

    path.append(path[0])
    distance = sum(dist_matrix[path[i]][path[i + 1]] for i in range(NUM_CITIES))
    all_paths.append(path)
    all_distances.append(distance)

    if distance < best_distance:
        best_distance = distance
        best_path = path

pheromone *= (1 - EVAPORATION)

for i in range(len(all_paths)):
    for j in range(NUM_CITIES):
        from_city = all_paths[i][j]
        to_city = all_paths[i][j + 1]
        pheromone[from_city][to_city] += Q / all_distances[i]
        pheromone[to_city][from_city] += Q / all_distances[i]

if iteration % 10 == 0 or iteration == NUM_ITERATIONS - 1:
    print(f"Iteration {iteration}: Best Distance = {best_distance:.2f}")

print("\nBest Path Found:")
print(" -> ".join(map(str, best_path)))
print(f"Total Distance: {best_distance:.2f}")
```

OUTPUT:

```
0 5 15 4
5 0 4 8
15 4 0 1
4 8 1 0
done

Input Initial Pheromone Matrix:
Enter the pheromone matrix row by row (space-separated). Type
'done' when finished:
0 4 10 3
4 0 1 2
10 1 0 1
3 2 1 0
done
Iteration 0: Best Distance = 14.00
Iteration 10: Best Distance = 14.00
Iteration 20: Best Distance = 14.00
Iteration 30: Best Distance = 14.00
Iteration 40: Best Distance = 14.00
Iteration 49: Best Distance = 14.00

Best Path Found:
3 -> 2 -> 1 -> 0 -> 3
Total Distance: 14.00
```

ANT COLONY OPTIMIZATION

- Pheromone
- Decision Making

$$\Delta\tau_{i,j}^{k} = \begin{cases} \frac{1}{L_k} & k^{th} \text{ ant travels on the edge } i,j \\ 0 & \text{otherwise} \end{cases}$$

$$\tau_{i,j}^{k} = \sum_{k=1}^{m} \Delta\tau_{i,j}^{k} \quad \text{without vaporization}$$

$$\tau_{i,j}^{k} = (1-\rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k} \quad \text{with vaporization}$$

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}$$

where : $\eta_{i,j} = \frac{1}{L_{i,j}}$

Algorithm

Steps:

1. Initialization:
   - Initialize pheromone levels τ_{ij} on all edges to a small positive constant
   - Initialize pheromone trails $\tau_{i,j} = \tau_0$ for all edge(i,j)
   - Compute heuristic information $\eta_{ij} = \frac{1}{D[i][j]}$ (Inverse of distance)

2. Repeat for each iteration $t = 1, 2, \ldots, T$.
   a. For each ant $k = 1, 2, \ldots, m$:
   - Place ant k on a randomly chosen starting city
   - Construct a tour by repeatedly selecting the next city according to a probabilistic rule

Bafna Gold
Date Page

For an ant currently at city i, the probability of moving to city j (not yet visited) is:

$$p_{ij}^{k} = \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\Sigma((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta})}$$

- Continue until all cities are visited to complete the tour

b. Evaluate the length of each ant's tour L_k.

3. Update pheromones:

a. Evaporation:

$$\tau_{i,j} = (1 - \rho) \cdot \tau_{ij}$$

b. Deposit pheromone

$$\tau_{ij} = \tau_{ij} + \Delta\tau_{ij}^{k}$$

where

$$\Delta\tau_{ij}^{k} = \frac{Q}{L_k}$$

4. Keep track of the best tour found so far.
5. After T iterations, output the best tour

ACO for TSP

```
import numpy as np
class ACO:
    def __init__(self, dist, ants, iters, decay, alpha=1, beta=
        self.dist = dist
        self.pher = np.ones(dist.shape) / len(dist)
        self.ants = ants
        self.iters = iters
        self.decay = decay
        self.alpha = alpha
        self.beta = beta
```

```
def run(self):
    best_path, best_len = None, float('inf')
    for _ in range.(self.iters):
        all_paths - [self.build_path() for _ in range(self.ants
        self.phes *= self.decay
        for path in all_paths:
            length = self.path_len(path)
            if length < best_len:
                best_path, best_len = path, length
            self.add_pheromone (path, length)
        print(f"Best length so far: {best_len:.2f}")
    return best_path, best_len
def build_path (self):
    path = [0]
    visited = set(path)
    for _ in range(len(self.dist)-1):
        current = path[-1]
        probs = self.probabilities (current, visited)
        next_city = np.random.choice (len(self.dist), p=probs)
        path.append (next_city)
        visited.add (next_city)
    return path
def add_pheromone (self, path, length):
    for i in range (len(path)):
        a, b = path[i], path[(i+1) % len(path)]
        self.pher[a, b] += 1 / length
        self.phes[b, a] += 1 / length
if __name__ == "__main__":
    dist = np.array([
        [0, 2, 2, 5, 7],
        [2, 0, 4, 8, 2],
        [2, 4, 0, 1, 3],
```

```
[5, 8, 1, 0, 2],
[7, 2, 3, 2, 0]
])
aco = ACO(dist, ants=10, iter=5, decay=0.5)
best_path, best_len = aco.run()
print("\n Best path:", best_path)
print(" Best length:", best_len)
```

Output:

Best length so far: 9.00
Best length so far: 9.00
Best length — — : 9.00
— " — : 9.00
— " — : 9.00

Best path: [0, 1, 4, 3, 2]
Best length: 9

Problem 5

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

```
import random
import math

weights = [10, 20, 30, 40, 15, 25, 35]
values = [60, 100, 120, 240, 80, 150, 200]
capacity = 100

n_items = len(weights)
```

```
n_nests = 15
max_iter = 50
pa = 0.25

def fitness(solution):
    total_weight = sum(w for w, s in zip(weights, solution) if s == 1)
    total_value = sum(v for v, s in zip(values, solution) if s == 1)
    if total_weight > capacity:
        return 0
    else:
        return total_value

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

def levy_flight(Lambda=1.5):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = random.gauss(0, sigma_u)
    v = random.gauss(0, 1)
    step = u / (abs(v) ** (1 / Lambda))
    return step

def get_cuckoo(nest, best_nest):
    new_nest = []
    for xi, bi in zip(nest, best_nest):
        step = levy_flight()
        val = xi + step * (xi - bi)

        s = 1 / (1 + math.exp(-val))
        new_val = 1 if s > 0.5 else 0
        new_nest.append(new_val)
    return new_nest

def cuckoo_search():
    nests = [generate_nest() for _ in range(n_nests)]
    fitness_values = [fitness(nest) for nest in nests]

    best_index = fitness_values.index(max(fitness_values))
    best_nest = nests[best_index][:]
    best_fitness = fitness_values[best_index]

    for iteration in range(1, max_iter + 1):
        for i in range(n_nests):
```

```python
            new_nest = get_cuckoo(nests[i], best_nest)
            new_fitness = fitness(new_nest)
            if new_fitness > fitness_values[i]:
                nests[i] = new_nest
                fitness_values[i] = new_fitness

        for i in range(n_nests):
            if random.random() < pa:
                nests[i] = generate_nest()
                fitness_values[i] = fitness(nests[i])

        current_best_index = fitness_values.index(max(fitness_values))
        current_best_fitness = fitness_values[current_best_index]

        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_index][:]

        if iteration % 10 == 0:
            print(f"Iteration {iteration}: Best value so far = {best_fitness}")

    return best_nest, best_fitness

if __name__ == "__main__":
    best_solution, best_value = cuckoo_search()
    total_weight = sum(w for w, s in zip(weights, best_solution) if s == 1)
    print(f"\nBest packing solution (1 = selected): {best_solution}")
    print(f"Total value of supplies packed: {best_value}")
    print(f"Total weight: {total_weight}")
```

OUTPUT:

```
Iteration 10: Best value so far = 570
Iteration 20: Best value so far = 570
Iteration 30: Best value so far = 590
Iteration 40: Best value so far = 590
Iteration 50: Best value so far = 590

Best packing solution (1 = selected): [0, 0, 0, 1, 0, 1, 1]
Total value of supplies packed: 590
Total weight: 100
```

Lab - 6

125

CUCKOO SEARCH ALGORITHM.

1. Set the initial value of the host nest size n; probability $P_a \in (0,1)$ and maximum number of iteration Maxt.
2. Set t := 0 {. Counter Initialization}.
3. For (i=1 : i≤n) do
4. Generate initial population of n host x_i^t.
5. Evaluate Fitness Function $f(x_i^t)$.
6. End for
7. Generate a new solution (cuckoo) x_i^{t+1} randomly by Levy Flight
8. Evaluate Fitness function x_i^{t+1} i.e., $f(x_i^{t+1})$
9. Choose a nest x_j among n solutions randomly.
10. If $(f(x_i^{t+1}) > f(x_j^t))$ then
11. Replace the solution x_j with the solution x_i^{t+1}
12. End if
13. Abandon a fraction Pa of worst nest
14. Build new nest at new location using Levy flight a fraction Pa of worse nest
15. Keep the best solution (nest with quality solution
16. Rank the solution and find current best solution
17. Set t = t + 1;
18. Until (t ≥ Maxt)
19. Produce the best solution.

```
import numpy as np
class CuckooSearchKnapsack:
    def _init_ (self, n_nests=25, pa=0.25, max_iter=500):
        self.n_nests = n_nests
        self.pa = pa
        self.max_iter = max_iter
        self.best_nest = None
```

```
    self.best_fitness = float('-inf')
    self.fitness_history = []
def fitness (solution):
    total_weight = np.sum(solution * weights)
    total_value = np. sum (solution * values)
    if total_weight > capacity:
        return 0
    else:
        return total_value
nests = np. random. randint(2, size=(n_nests, n_items))
fitnesses = np. array([fitness(n) for n in nests])
best_idx = np. argmax (fitnesses)
best_nest = nests [best_idx]. copy()
best_fitness = fitnesses[best_idx]
print(f" Initial best fitness: {best_fitness}")

def bit flip_mutation (solution, mutation_rate=0.3):
    new_sol = solution. copy( )
    for i in range (len(solution)):
        if np. random. rand()< mutation_rate:
            new-sol [i] = 1 - new_sol[i]
    return new-sol
for iteration in range (max_iter):
    new_nests = np. array([flip_mutation(nests [i])
     for i in range (n_nests)])
    new_fitnesses = np. array ([fitness (n) for n in
     new_nests])
    for i in range (n_nests):
        if new_fitnesses[i] > fitnesses[i]:
        nests [i]= new_nests [i]
        fitnesses[i]= new_fitnesses [i]
```

```
current_best_idx = np.argmax(fitnesses)
    if fitnesses[current_best_idx] > best_fitness:
        best_fitness = fitnesses[current_best_idx]
        best_nest = nests[current_best_idx].copy()

    worst_idx = np.argmin(fitnesses)
    if np.random.rand() < pa:
        nests[worst_idx] = np.random.randint(2, size=n_items)
        fitnesses[worst_idx] = fitness(nests[worst_idx])
    print(f"Iter {iteration+1}: Best fitness = {best_fitness}")
print("\n Best solution found:")
print(f"Items taken: {best_nest}")
print(f"Total weight: {np.sum(best_nest * weights)}")
print(f"Total value: {best_fitness}")
```

Output:

```
Initial best fitness: 9.
Iter 1: Best fitness = 10.
Iter 2: Best fitness = 10.
Iter 3: Best fitness = 10
Iter 4: Best fitness = 10
Iter 5: Best fitness = 10

Best Solution found:
Items taken: [0 1 0 1]
Total weight: 8
Total value: 10
```

10/10

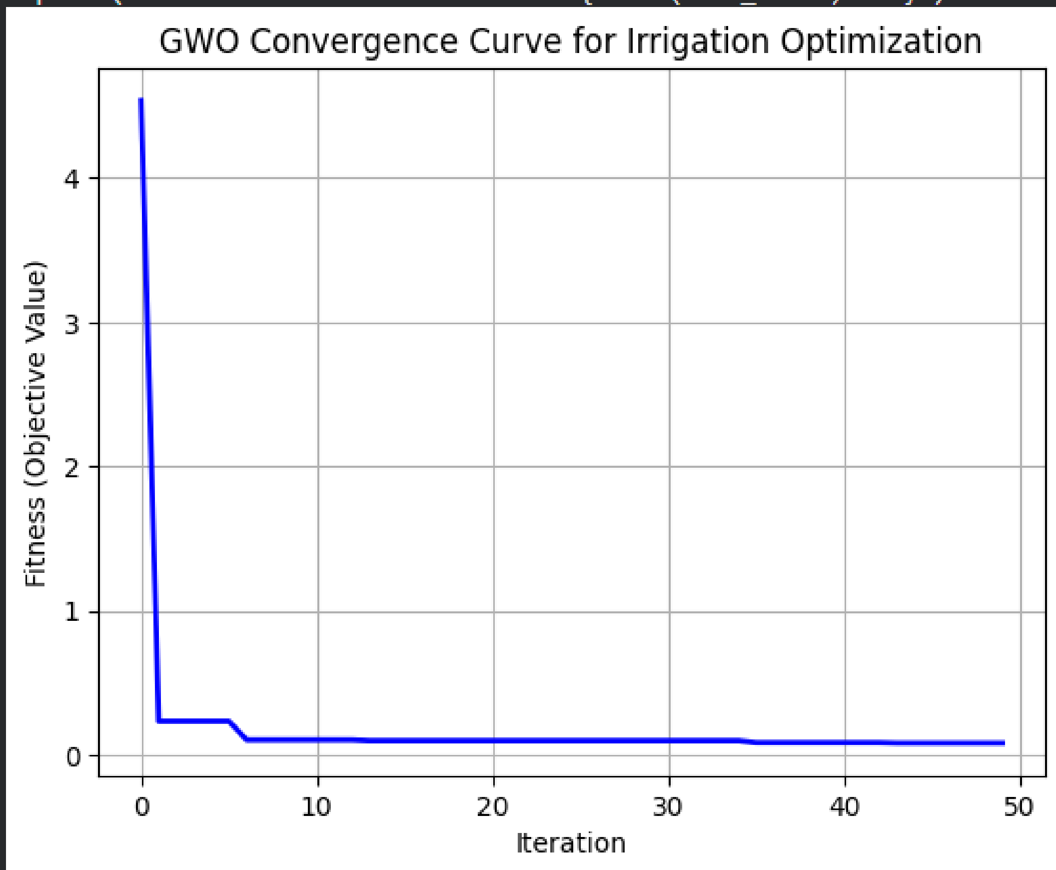
Program 6

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.

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

def irrigation_objective(x):
    # Parameters for irrigation problem
    a = 0.8      # yield penalty factor
    b = 5        # water cost factor
    x_opt = 60   # optimal irrigation level (e.g., mm/day)
    return a * (x - x_opt)**2 + b / x

def GWO(obj_func, lb, ub, dim, n_wolves, max_iter):
    # Initialize wolves randomly within bounds
    wolves = np.random.uniform(lb, ub, (n_wolves, dim))
    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    convergence_curve = []

    for t in range(max_iter):
        for i in range(n_wolves):
            fitness = obj_func(wolves[i])
            # Update alpha, beta, delta
            if fitness < alpha_score:
                alpha_score, alpha = fitness, wolves[i].copy()
            elif fitness < beta_score:
                beta_score, beta = fitness, wolves[i].copy()
            elif fitness < delta_score:
                delta_score, delta = fitness, wolves[i].copy()

        # Linearly decreasing 'a' from 2 to 0
        a = 2 - t * (2 / max_iter)
```

```
        # Update positions of wolves
        for i in range(n_wolves):
            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

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

        convergence_curve.append(alpha_score)

    return alpha, alpha_score, convergence_curve

dim = 1
lb, ub = 10, 100     # irrigation limits (mm/day)
n_wolves = 20
max_iter = 50

best_pos, best_score, curve = GWO(irrigation_objective, lb, ub, dim, n_wolves, max_iter)
print(f"Optimal Irrigation Level: {best_pos[0]:.3f} mm/day")
print(f"Minimum Cost Function Value: {float(best_score):.4f}")

plt.plot(curve, 'b-', linewidth=2)
plt.title('GWO Convergence Curve for Irrigation Optimization')
plt.xlabel('Iteration')
plt.ylabel('Fitness (Objective Value)')
plt.grid(True)
plt.show()
```
OUTPUT:

```
Optimal Irrigation Level: 59.990 mm/day
Minimum Cost Function Value: 0.0834
/tmp/ipython-input-2606069732.py:76: DeprecationWarning: Conversion of an array w
  print(f"Minimum Cost Function Value: {float(best_score):.4f}")
```



Lab 7

Grey Wolf Optimizer

Initialize population of wolves
Evaluate fitness of each wolf
Identify alpha, beta, and delta wolves
Update positions of wolves
Handle boundaries
Repeat until max iterations or convergence
Return alpha wolf as best solution.

Environmental & Agricultural system

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

def irrigation_objective(x):
 a = 0.8
 b = 5
 x_opt = 60
 return a * (x - x_opt)**2 + b/x
def GWO(obj_func, lb, ub, dim, n_wolves, max_iter):
    wolves = np.random.uniform(lb, ub, (n_wolves, dim))
    alpha, beta, delta = np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = float("inf"), float.
    ("inf"), float("inf")
    convergence_curve = [ ]
    for t in range(max_iter):
        for i in range(n_wolves):
            fitness = obj_fun(wolves[i])
            if fitness < alpha_score:
                alpha_score, alpha = fitness, wolves[i].copy()
            elif fitness < beta_score:
                beta_score, beta = fitness, wolves[i].copy()
            elif fitness < delta_score:
```

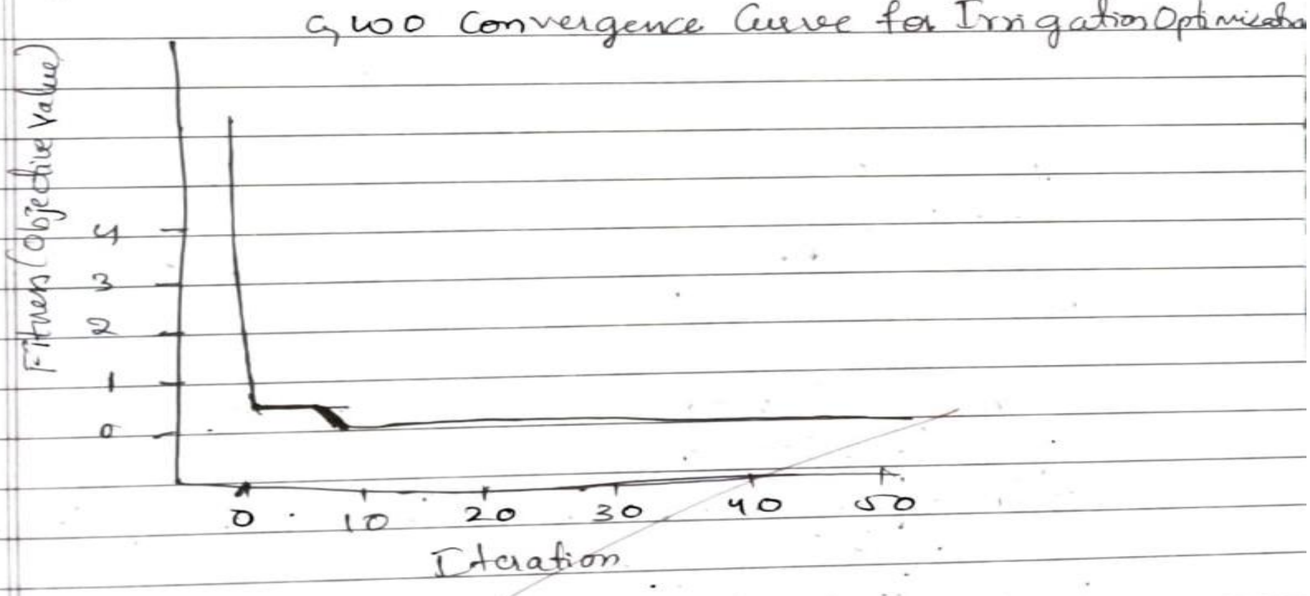
```
    delta_score, delta = fitness, wolves[i].copy()
a = 2 - t * (2/max_iter)
for i in range(n_wolves):
    for j in range(dim):
        r1, r2 = np.random.rand(), np.random.rand
        A1, C1 = 2* a * r1 - a, 2 * r2
        D_alpha = abs(C1 * alpha[j] - wolves[i][j])
        x1 = alpha[j] - A1 * D_alpha
        r1, r2 = np.random.rand(), np.random.rand()
        A2, C2 = 2 * a * r1 - a, 2 * r2
        D_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
    wolves[i] = np.clip(wolves[i], lb, ub)
  convergence_curve.append(alpha_score)
 return alpha, alpha_score, convergence_curve
dim = 1
lb, ub = 10, 100
n_wolves = 20
max_iter = 50
best_pos, best_score, curve = GWO(irrigation_objective, lb,
 ub, dim, n_wolves, max_iter)
```

```
print(f"Optimal Irrigation Level: {best_pos[0]:.3f}mm/
      day")
print(f"Minimum Cost Function Value: {float(best_score):.
      4f}")
plt.plot(curve, 'b-', linewidth=2)
plt.title('GWO Convergence Curve for Irrigation Optimization')
plt.xlabel('Iteration')
plt.ylabel('Fitness (Objective Value)')
plt.grid(True)
plt.show()
```

Output:

Optimal Irrigation Level: 59.990mm/day

Minimum Cost Function Value: 0.0834



6/11

Program 7

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.

```
#PCA
import numpy as np
import random

num_customers = int(input("Enter number of customers (excluding depot): "))
num_vehicles = int(input("Enter number of vehicles: "))

print("\nEnter the distance matrix (including depot 0):")
print(f"Matrix should be {num_customers + 1} x {num_customers + 1}")
distance_matrix = []

for i in range(num_customers + 1):
    row = list(map(int, input(f"Row {i+1}: ").split()))
    distance_matrix.append(row)

distance_matrix = np.array(distance_matrix)

rows = int(input("\nEnter number of grid rows: "))
cols = int(input("Enter number of grid columns: "))
grid_dim = (rows, cols)
population_size = rows * cols

num_generations = int(input("\nEnter number of generations: "))

def generate_individual():
    perm = list(range(1, num_customers + 1))
    random.shuffle(perm)
    return perm

population = [generate_individual() for _ in range(population_size)]

def fitness(individual):
```

```
    split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
    total_distance = 0
    for i in range(num_vehicles):
        route = [0] + individual[split_points[i]:split_points[i+1]] + [0]
        for j in range(len(route) - 1):
            total_distance += distance_matrix[route[j], route[j+1]]
    return total_distance

def get_neighbors(idx):
    r, c = divmod(idx, grid_dim[1])
    neighbors = []
    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < grid_dim[0] and 0 <= nc < grid_dim[1]:
                n_idx = nr * grid_dim[1] + nc
                if n_idx != idx:
                    neighbors.append(n_idx)
    return neighbors

def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[a:b] = parent1[a:b]

    pointer = b
    for gene in parent2[b:] + parent2[:b]:
        if gene not in child:
            if pointer == size:
                pointer = 0
            child[pointer] = gene
            pointer += 1
    return child

def mutate(individual):
    a, b = random.sample(range(len(individual)), 2)
    individual[a], individual[b] = individual[b], individual[a]
    return individual

def pca_iteration(pop):
    new_pop = pop.copy()
    for idx in range(len(pop)):
        neighbors = get_neighbors(idx)
```

```
        partner_idx = random.choice(neighbors)
        parent1 = pop[idx]
        parent2 = pop[partner_idx]

        child = crossover(parent1, parent2)
        if random.random() < 0.2:
            child = mutate(child)

        if fitness(child) < fitness(pop[idx]):
            new_pop[idx] = child
    return new_pop

for gen in range(num_generations):
    population = pca_iteration(population)
    best_fitness = min(fitness(ind) for ind in population)
    print(f"Generation {gen+1}: Best total distance = {best_fitness}")

best_individual = min(population, key=fitness)
print("\nBest route assignment (split evenly):")
split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
for i in range(num_vehicles):
    route = [0] + best_individual[split_points[i]:split_points[i+1]] + [0]
    print(f"Vehicle {i+1} route: {route}")
print(f"Total distance: {fitness(best_individual)}")
```

OUTPUT:

```
Enter number of customers (excluding depot): 3
Enter number of vehicles: 2

Enter the distance matrix (including depot 0):
Matrix should be 4 x 4
Row 1: 0 2 9 10
Row 2: 2 0 6 4
Row 3: 9 6 0 8
Row 4: 10 4 8 0

Enter number of grid rows: 3
Enter number of grid columns: 3

Enter number of generations: 10
Generation 1: Best total distance = 31
Generation 2: Best total distance = 31
Generation 3: Best total distance = 31
Generation 4: Best total distance = 31
Generation 5: Best total distance = 31
Generation 6: Best total distance = 31
Generation 7: Best total distance = 31
Generation 8: Best total distance = 31
Generation 9: Best total distance = 31
Generation 10: Best total distance = 31

Best route assignment (split evenly):
Vehicle 1 route: [0, 1, 0]
Vehicle 2 route: [0, 2, 3, 0]
Total distance: 31
```

Parallel Cellular Algorithm

Procedure
Define the Problem
Example Problem: Minimize
$f(x) = x^2 - 4x + 4$

Goal: Find the value of x that minimizes f(x)

Initialize Parameters
- Number of Cells: 100 cells in the grid
- Grid Size: 2D grid (10x10)
- Neighborhood Structure: 3x3 neighborhood
- Iterations: 100 iterations

```
import numpy as np
import matplotlib.pyplot as plt
from multiprocessing im
cpu_count
def f(x,y):
    return x**2 + y**2+10*np.sin(x) +7*np.cos(y)
num_cells =100
iteration = 5
x_range =[-10,10]
y_range =[-10,10]
cells = np.random.uniform( low =[x_range[0], y_
  range [0]],
high =[x_range[1], y_range[1]],
size =(num_cells, 2))
def evaluate(cell):
  x,y = cell
  return f(x,y)
```

```
def parallel_fitness(cells):
    with Pool(cpu_count()) as p:
        fitness = p.starmap(f, [(x, y) for x, y in
          cells])
    return np.array(fitness)
best_solutions = []
for it in range(iterations):
    fitness = parallel_fitness(cells)
    best_idx = np.argmin(fitness)
    best_cell = cells[best_idx]
 best_solutions.append[fitness[best_idx]]
    for i in range(num_cells):
      neighbor = cells[np.random.randint(0, num_
          cells)]
        cells[i] += 0.1 * (neighbor - cells[i])
        cells[i] += np.random.uniform(-0.1, 0.1, size=2)
    cells = np.clip(cells, [x_range[0], y_range[0]],
                 [x_range[1], y_range[1]])
best_x, best_y = cells[best_idx]
print(f" Best Solution found: x = {best_x:.4f}, y=
   {best_y:.4f}, f(x,y) = {f(best_x, best_y):.4f}")
plt.plot(best_solutions)
plt.title("Parallel Cellular Algorithm-Function
             Optimization")
plt.xlabel("Iteration")
plt.ylabel("Best Fitness Value")
plt.grid(True)
plt.show()
```

Output

Best solution found: x = 0.0632, y = 0.6227, f(x,y) =
6.7094

Parallel Cellular Algorithm - Function Optimization