

# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



## LAB RECORD

**Bio Inspired Systems (23CS5BSBIS)** *Submitted by*

**TARUN S SUNADOLI (1WA23CS015)**

*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 **Tarun S Sunadoli (1WA23CS015)**, who is a bonafide student of **B.M.S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Dr. Shwetha K S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--------------------------------------------------------------------	------------------------------------------------------------------

# Index

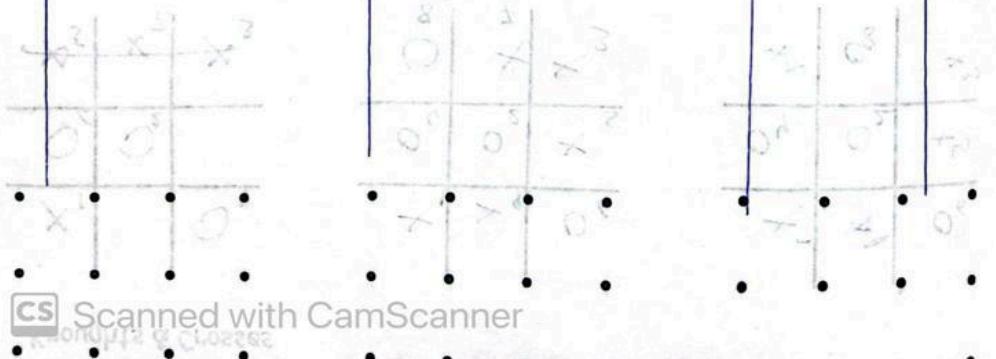
Sl. No.	Date	Experiment Title	Page No.
1	21/08/25	Genetic Algorithms	
2	28/08/25	Gene Expression Algorithm	
3	11/09/25	Particle Swarm Optimization	
4	09/10/25	Ant Colony Optimization	
5	16/10/25	Cuckoo Search	
6	06/11/25	Grey Wolf Optimizer	
7	13/11/25	Parallel Cellular Algorithms	

**Github Link:**

<https://github.com/Tarun-619/BIS-lab-1WA23CS015>

Iarun S Sunaddi		IWA23CS015	V-G	
S.no	Date	Topic	Pg no. marks	Signature
1	14/08/25	LAB-1 (BFS)	10/10	
2	21/08/25	LAB-2 (Genetic algo)	10/10	
3	28/08/25	LAB-3 (Gene exp)	10/10	AS
4	11/09/25	LAB-4 (PSO)	10/10	AS
5	9/10/25	LAB-5 (ACO)	10/10	AS
6	16/10/25	LAB-6 (CSA)	10/10	AS
7	6/11/25	LAB-7 (GWO)	10/10	AS
8	13/11/25	LAB-8 (PCA)	10/10	AS

(10)  
(10) *AS 2024*




## LAB 1 ➔ BIO-INSPIRED SYSTEM ➔

- 1) Genetic Algorithm for optimization problems  
→ A genetic algo (GA) is a technique where computers search for better sol's, by imitating the process of natural evolution. It begins with a group of random possible sol's, which are tested for their performance and the parent are then combined using crossover to form new solutions and sometimes small mutations are added to maintain variety. By repeating this process many times, the overall quality of sol's improve step by step.

### Applications:

- Solving optimization tasks
- Machine learning techniques
- Engineering & design problems
- Game strategies
- Bioinformatics research
- Robotic development
- Financial planning
- Creative and artistic applications.

### 2) Ant Colony Optimization:

- Vehicle routing problem
- Job scheduling
- Network routing

### 3) Cuckoo search:

- Optimizing weights of NN
- Clustering
- Job scheduling

### 4) Particle Swarm Optimization:

- Neural network
- Clustering
- Feature select,

### 5) Parallel Cellular organism:

- Image processing
- Finance
- Environment modeling

### 6) Optimization is a Gene exp. algo

- Engineering design
- AI
- Finance

Scanned with CamScanner

Scanned with CamScanner

## 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.

Algorithm:

L A B - 2						
Genetic algorithm,						
- the initial population is considered for the given values of n, ranging from 0 to 31						
- the initial values are 12, 25, 5, 19						
<u>Steps</u>						
(1) Select initial population						
(2) Calculate the fitness						
$\text{prob} = f(n)$						
$\text{expected count} = \frac{f(n)}{\text{avg } (\Sigma f(n))}$						
(3) Select mating pool						
(4) Crossover						
(5) Mutation						
# String no. Initial population n fitness prob $f(n)=n^2$ popularity Expected outcome						
1	01100	12	144	0.1248	12.48	0.4987
2	11001	25	625	0.5411	54.11	2.1643
3	00101	5	25	0.0216	2.16	0.0866
4	10011	19	361	0.3126	31.26	1.25
	sum	1155				
	avg	288.75				
	max	625				
# Selecting mating pool						
String no. mating pool cross over offspr. final fitness $f(n)=n$						
1	01100	7	01101	13	169	
2	11001	3	11000	24	576	
3	11001	2	11011	27	729	
4	10011	2	10001	17	289	
	sum					
	avg					
	max	729				

#	mutation	String	offspring	mutation	dual	fitness
1	01101	10000	11101	29	841	
2	11000	00000	11000	24	576	
3	11011	00000	11011	27	729	
4	10001	00101	10100	20	400	

Import random

CHROM\_LENGTH = 5

CROSS\_RATE = 0.8

MUT RATE = 0.1

def fitness(n):  
 return n \* 2

def binary\_decode(b):  
 return int(b, 2)

def encode(n):  
 return format(n, f'0{CHROM\_LENGTH}b')

def roulette\_selection(pop, fitness):  
 total\_fit = sum(fitness)  
 pick = random.uniform(0, total\_fit)  
 curr=0  
 for f in enumerate(fitness)  
 curr+=f  
 if curr > pick:  
 return pop[f[0]]  
 return pop[-1]

CS Scanned with CamScanner

prob = [f / total\_fit for f in fitness]  
expected = [p + possible\_POP\_SIZE for p in prob]

print(f'{1}/{generations}'")

for i in range(POP\_SIZE):  
 print(f'{i}={decoded[i]}', bin=(population[i]))  
 fit = fitness[i], prob = (prob[i] \* 3) \*\* 2  
 exp\_count = expected[i] = 2 \*\* 2  
  
 new\_pop = []  
 while len(new\_pop) < POP\_SIZE:  
 p1 = roulette\_selection(population, fitness)  
 p2 = roulette\_selection(population, fitness)  
 c1, c2 = crossover(p1, p2)  
 c1, c2 = mutate(c1), mutate(c2)  
 new\_pop.append((c1, c2))

population = new\_pop[:POP\_SIZE]

decoded = [decoded(c) for c in population]  
fitness = fitness([m for m in decoded])  
best\_idx = fitness.index(max(fitness))  
print(f'{1}/{generations}: {decoded[best\_idx]}, population[{best\_idx}], fitness[{best\_idx}]')

genetic algorithm()

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_ip = input("Enter initial population value (space separated, eg 12 35 4): ")
```

values = list(map(int, user\_ip.split(' ')))  
gen = int(input("Enter no. of generations: "))

POP\_SIZE = len(values)

population = [encode(n) for n in values]  
print(f'{0}/{gen} Initial population: {population}, (decode(c) for c in population)')

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

total\_fit = sum(fitness)

output

enter init. population values (eg. 1 3 4 5): 3 6 7 2  
num of generations: 5

initial population: ['00011', '00110', '00111', '00010']  
[3, 6, 7, 2]

gen 1:  
chrom 1: 00011, decoded = 3, fitness = 9, prob = 0.05  
chrom 2: 00110, decoded = 6, fitness = 36, prob = 0.36  
chrom 3: 00111, decoded = 7, fitness = 49, prob = 0.38  
chrom 4: 00010, decoded = 2, fitness = 4, prob = 0.04

gen 2:  
chrom 1: 10111, decoded = 23, fitness = 519, prob = 0.866  
chrom 2: 00011, decoded = 3, " = 9, " = 0.015  
chrom 3: 00111, decoded = 7, " = 49, " = 0.081  
chrom 4: 01010, decoded = 5, " = 25, " = 0.04

gen 3:  
chrom 1: 10011, decoded = 19, fitness = 361, prob = 0.20  
" 2: 10111, " = 23, " = 519, " = 0.29  
" 3: 10111, " = 23, " = 519, " = 0.29  
" 4: 10011, " = 19, " = 361, " = 0.20

gen 4:  
chrom 1: 10011, decoded = 19, fitness = 361, prob = 0.23  
" 2: 10111, " = 23, " = 519, " = 0.27  
" 3: 10011, " = 19, " = 361, " = 0.22  
" 4: 00111, " = 7, " = 49, " = 0.03

gen 5:  
chrom 1: 10111, decoded = 23, fitness = 529, prob = 0.272  
" 2: 10111, " = 23, " = 529, " = 0.27  
" 3: 10011, " = 19, " = 361, " = 0.18  
" 4: 10111, " = 23, " = 529, " = 0.272

CS Scanned with CamScanner

final best soln:  
 chromosome: 10111 decoded: 23 fitness: 529  
~~10111~~

Code:

```

import random

CHROM_LENGTH = 5
CROSS_RATE = 0.8
MUT_RATE = 0.1

def fitness(n):
    return n ** 2

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

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

def roulette_selection(pop, fitness_values):
    total_fit = sum(fitness_values)
    pick = random.uniform(0, total_fit)
    curr = 0
    for i, f in enumerate(fitness_values):
        curr += f
        if curr > 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 j in range(CHROM_LENGTH):
        if random.random() < MUT_RATE:
            chrom_list[j] = '1' if chrom_list[j] == '0' else '0'
    return ''.join(chrom_list)

def genetic_algorithm():
    user_ip = input("enter ini pop values (e.g. 1 2 3 4): ")
    values = list(map(int, user_ip.split()))
    generations = int(input("enter num of gen: "))

```

```

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

print("\ini pop:", population, [decode(c) for c in population])

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

    total_fit = sum(fitness_values)
    probs = [f/total_fit for f in fitness_values]

    print(f"\gen {gen}:")
    for i in range(POP_SIZE):
        print(f"chromosome {i+1}: {population[i]}, decoded={decoded[i]}, "
              f"fitness={fitness_values[i]}, prob={probs[i]:.3f}")

    new_pop = []
    while len(new_pop) < POP_SIZE:
        p1 = roulette_selection(population, fitness_values)
        p2 = roulette_selection(population, fitness_values)
        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]
    fitness_values = [fitness(n) for n in decoded]
    best_idx = fitness_values.index(max(fitness_values))
    print("\final soln:")
    print("chromosome:", population[best_idx],
          "decoded:", decoded[best_idx],
          "fitness:", fitness_values[best_idx])

genetic_algorithm()

```

op:

---

```
enter ini pop values (e.g. 1 2 3 4): 3 6 7 2
enter num of gen: 5
\ini pop: ['00011', '00110', '00111', '00010'] [3, 6, 7, 2]
\gen 1:
chromosome 1: 00011, decoded=3, fitness=9, prob=0.092
chromosome 2: 00110, decoded=6, fitness=36, prob=0.367
chromosome 3: 00111, decoded=7, fitness=49, prob=0.500
chromosome 4: 00010, decoded=2, fitness=4, prob=0.041
\gen 2:
chromosome 1: 10111, decoded=23, fitness=529, prob=0.864
chromosome 2: 00011, decoded=3, fitness=9, prob=0.015
chromosome 3: 00111, decoded=7, fitness=49, prob=0.080
chromosome 4: 00101, decoded=5, fitness=25, prob=0.041
\gen 3:
chromosome 1: 10011, decoded=19, fitness=361, prob=0.203
chromosome 2: 10111, decoded=23, fitness=529, prob=0.297
chromosome 3: 10111, decoded=23, fitness=529, prob=0.297
chromosome 4: 10011, decoded=19, fitness=361, prob=0.203
\gen 4:
chromosome 1: 10011, decoded=19, fitness=361, prob=0.278
chromosome 2: 10111, decoded=23, fitness=529, prob=0.407
chromosome 3: 10011, decoded=19, fitness=361, prob=0.278
chromosome 4: 00111, decoded=7, fitness=49, prob=0.038
\gen 5:
chromosome 1: 10111, decoded=23, fitness=529, prob=0.272
chromosome 2: 10111, decoded=23, fitness=529, prob=0.272
chromosome 3: 10011, decoded=19, fitness=361, prob=0.185
chromosome 4: 10111, decoded=23, fitness=529, prob=0.272
inal soln:
chromosome: 10111 decoded: 23 fitness: 529
```

## Program 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.

Algorithm:

LAB - 3

28/8/2025

<p>Optimization via Gene Expr. Algo. + Import random def fft(c):     return -(10 + (c * 2 - 10) * math.cos(2 * math.pi * c)) def encode(n):     return format(n, f'{0:0{chrom LENGTH}b}') def decode(b):     return int(b, 2)  def gene_exp(chrom):     return decode(chrom)  def select(pop, fft):     s, r, c = sum(ft), random.uniform(0, sum(ft)), 0     for i, f in enumerate(ft):         e += f         if e &gt; r:             return pop[i]  def crossover(p1, p2):     pt = random.randint(1, 2 - 1)     return p1[:pt] + p2[pt:], p2[:pt] + p1[pt:]  def mutate(c):     return ''.join('1' if (b == '0' and random.random() &lt; 0.1)                   else '0' if (b == '1' and random.random() &lt; 0.1)                   else b for b in c)</p> <td><p>Output Initial Population ['01100', '10111', '00101', '10011'] [12, 23, 5, 19]</p><p>Generation 1: n = 12, bin = 01100, fft = -14.0, prob = 0.136, exp_count = 1.00 n = 23, bin = 10111, fft = -5.0, prob = 0.500, exp_count = 2.00 n = 5, bin = 00101, fft = -25.0, prob = 0.024, exp_count = 0.00 n = 19, bin = 10011, fft = -36.0, prob = 0.341, exp_count = 1.00 Final Best Solution: 4 00100 fft: -16.0</p><p style="text-align: center;">X</p></td>	<p>Output Initial Population ['01100', '10111', '00101', '10011'] [12, 23, 5, 19]</p> <p>Generation 1: n = 12, bin = 01100, fft = -14.0, prob = 0.136, exp_count = 1.00 n = 23, bin = 10111, fft = -5.0, prob = 0.500, exp_count = 2.00 n = 5, bin = 00101, fft = -25.0, prob = 0.024, exp_count = 0.00 n = 19, bin = 10011, fft = -36.0, prob = 0.341, exp_count = 1.00 Final Best Solution: 4 00100 fft: -16.0</p> <p style="text-align: center;">X</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Code:

```
import random
import math

CHROM_LENGTH = 5
POP_SIZE = 4
CROSS_RATE = 0.8
MUT_RATE = 0.1
GENERATIONS = 10

def fitness(x):
    return -(10 + (x**2 - 10 * math.cos(2 * math.pi * x)))

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

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

def gene_expression(chrom):
    return decode(chrom)

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 gene_expression_algorithm():
    population = [encode(x) for x in [7, 15, 22, 28]]
    print("Initial Population:", population, [decode(c) for c in population])

    for gen in range(1, GENERATIONS + 1):
        expressed = [gene_expression(c) for c in population]
        fitnesses = [fitness(x) for x in expressed]
        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={expressed[i]}, bin={population[i]}, fit={fitnesses[i]:.3f}, "
                  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]

expressed = [gene_expression(c) for c in population]
fitnesses = [fitness(x) for x in expressed]
best_idx = fitnesses.index(max(fitnesses))
print("\nfinal best sol:", expressed[best_idx], population[best_idx], "fitness=",
fitnesses[best_idx])

gene_expression_algorithm()

```

Op:

Initial Population: ['00111', '01111', '10110', '11100'] [7, 15, 22, 28]

Generation 1

x=7, bin=00111, fit=-49.000, prob=0.032, exp\_count=0.13  
x=15, bin=01111, fit=-225.000, prob=0.146, exp\_count=0.58  
x=22, bin=10110, fit=-484.000, prob=0.314, exp\_count=1.26  
x=28, bin=11100, fit=-784.000, prob=0.508, exp\_count=2.03

Generation 2

x=14, bin=01110, fit=-196.000, prob=0.647, exp\_count=2.59  
x=7, bin=00111, fit=-49.000, prob=0.162, exp\_count=0.65  
x=3, bin=00011, fit=-9.000, prob=0.030, exp\_count=0.12  
x=7, bin=00111, fit=-49.000, prob=0.162, exp\_count=0.65

Generation 3

x=7, bin=00111, fit=-49.000, prob=0.094, exp\_count=0.37  
x=5, bin=00101, fit=-25.000, prob=0.048, exp\_count=0.19  
x=3, bin=00011, fit=-9.000, prob=0.017, exp\_count=0.07  
x=21, bin=10101, fit=-441.000, prob=0.842, exp\_count=3.37

Generation 4

x=6, bin=00110, fit=-36.000, prob=0.114, exp\_count=0.45  
x=14, bin=01110, fit=-196.000, prob=0.618, exp\_count=2.47  
x=6, bin=00110, fit=-36.000, prob=0.114, exp\_count=0.45  
x=7, bin=00111, fit=-49.000, prob=0.155, exp\_count=0.62

Generation 5

x=6, bin=00110, fit=-36.000, prob=0.290, exp\_count=1.16  
x=6, bin=00110, fit=-36.000, prob=0.290, exp\_count=1.16  
x=6, bin=00110, fit=-36.000, prob=0.290, exp\_count=1.16  
x=4, bin=00100, fit=-16.000, prob=0.129, exp\_count=0.52

Generation 6

x=6, bin=00110, fit=-36.000, prob=0.127, exp\_count=0.51  
x=6, bin=00110, fit=-36.000, prob=0.127, exp\_count=0.51  
x=4, bin=00100, fit=-16.000, prob=0.056, exp\_count=0.23  
x=14, bin=01110, fit=-196.000, prob=0.690, exp\_count=2.76

Generation 7

x=14, bin=01110, fit=-196.000, prob=0.690, exp\_count=2.76  
x=4, bin=00100, fit=-16.000, prob=0.056, exp\_count=0.23  
x=6, bin=00110, fit=-36.000, prob=0.127, exp\_count=0.51  
x=6, bin=00110, fit=-36.000, prob=0.127, exp\_count=0.51

Generation 8

x=6, bin=00110, fit=-36.000, prob=0.082, exp\_count=0.33  
x=15, bin=01111, fit=-225.000, prob=0.510, exp\_count=2.04  
x=6, bin=00110, fit=-36.000, prob=0.082, exp\_count=0.33  
x=12, bin=01100, fit=-144.000, prob=0.327, exp\_count=1.31

Generation 9

x=22, bin=10110, fit=-484.000, prob=0.864, exp\_count=3.46  
x=2, bin=00010, fit=-4.000, prob=0.007, exp\_count=0.03  
x=6, bin=00110, fit=-36.000, prob=0.064, exp\_count=0.26  
x=6, bin=00110, fit=-36.000, prob=0.064, exp\_count=0.26

Generation 10

x=7, bin=00111, fit=-49.000, prob=0.312, exp\_count=1.25  
x=6, bin=00110, fit=-36.000, prob=0.229, exp\_count=0.92  
x=6, bin=00110, fit=-36.000, prob=0.229, exp\_count=0.92  
x=6, bin=00110, fit=-36.000, prob=0.229, exp\_count=0.92

final best sol: 6 00110 fitness= -36.0

### Program 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.

Algorithm:

LAB-4 PSO	
<u>Application &amp; Disease detection and classification</u>	
→ suppose we have patient data	
• features → (pp)	
- BP → $x_1$	
- Cholesterol → $x_2$	
- Sugar level → $x_3$	
- Age → $x_4$	
$p_1 = [1, 0, 1, 0]$	
• result (op)	
- 1 → Disease	
- 0 → Healthy	
<u>Feature selection</u>	
- medical datasets often have many features like above mentioned	
- we need search for the best subsets of features	
<u>Classifier</u>	
- PSO parameters	
<u>Fitness function</u>	
- f1 score, balance acc, precision & recall	
- error rate (MSE, cross-entropy)	
<del><math>f = 1 - \text{classification accuracy}</math></del>	
<u>Algo</u>	
- initialize particles	
- evaluate classifier using selected parameters	
- calculate fitness	
- pbest and gbest	
- update position and velocity	
- repeat until convergence	

Patient	BP(m <sub>1</sub> )	SL(m <sub>2</sub> )	Class(y)
1	130	200	1 (disease)
2	120	150	0 (Healthy)
3	140	220	1
4	110	130	0

goal: to classify patient healthy or disease

we have:

$$v_p^{t+1} = v_p^t + c_1 \tau_1 (pb_{p,t} - p_p^t) + c_2 \tau_2 (gb^t - p_p^t)$$

↓  
diversification      ↓  
Intensification

consider,

$$p_1 = [1, 0]$$

$$p_2 = [0, 1]$$

both having acc & g.best = [1, 0]

$$v_i(t+1) = w \cdot v_i(t) + c_1 \tau_1 (pb_{best,i} - pos_i) + c_2 \tau_2 (gbest - pos_i)$$

consider values for  $w, c_1, c_2, \tau_1, \tau_2$

def sigmoid(v):

$$\text{return } 1 / (1 + \exp(-v))$$

def fitness\_func(mask):

if no feature (mask):  
return 1.0

$$acc = classifier.acc(x[T], mask[J], y)$$

return 1 - acc

EX: if [1, 0, 1, 0] → ↓ Selected  
0 - Ignored

pseudocode:

```

for t in range (max_iter):
    for i in range (num_particles):
        r1, r2 = random(), random()
        velocity[i] = (
            w * velocity[i] +
            c1 * r1 * (pbest[i] - pos[i]) +
            c2 * r2 * (gbest - pos[i])
        )
    
```

$$prob = \text{sigmoid}(velocity[i])$$

$$pos[i] = [1] \text{ if random()} < p \text{ else } 0 \text{ for } p < prob$$

$$\text{score} = \text{fitness funct}(pos[i])$$

if score < pbest[i]:

$$pbest[i] = score$$

$$pbest[i] = pos[i].copy()$$

$$gbest\_pos = \text{argmin}(pbest\_val)$$

if pbest\_val[gbest\_pos] < fitness\_func(gbest):

$$gbest = pbest[gbest\_pos].copy()$$

print("Iteration", t, "Best acc", 1 - fitness\_func(gbest))

return gbest\_pos, 1 - fitness\_func(gbest\_pos)

OPP

Iteration 0: Best acc = 0.6000

Iteration 1: Best acc = 0.8160

" 15     " = 0.8750

" 20     " = 0.8750

Iteration 2: Best feature subset (gbest) = [1, 0, 1, 1, 2]

Code: import random

```

def profit_function(x):
    return -0.5 * (x**2) + 20 * x - 30

class Particle:
    def __init__(self, lower_bound, upper_bound):
        self.position = random.uniform(lower_bound, upper_bound)
        self.velocity = random.uniform(-1, 1)
        self.pbest_position = self.position
        self.pbest_value = profit_function(self.position)

class PSO:
    def __call__(self, num_particles=10, max_iterations=10, lower_bound=0, upper_bound=50,
    w=0.5, c1=1.5, c2=1.5):
        swarm = [Particle(lower_bound, upper_bound) for _ in range(num_particles)]
        gbest_position = swarm[0].position
        gbest_value = swarm[0].pbest_value

        for particle in swarm:
            if particle.pbest_value > gbest_value:

```

```

        gbest_value = particle.pbest_value
        gbest_position = particle.pbest_position

    for iteration in range(max_iterations):
        for particle in swarm:
            profit = profit_function(particle.position)
            if profit > particle.pbest_value:
                particle.pbest_value = profit
                particle.pbest_position = particle.position
            if profit > gbest_value:
                gbest_value = profit
                gbest_position = particle.position
        for particle in swarm:
            r1 = random.random()
            r2 = random.random()
            particle.velocity = (
                w * particle.velocity
                + c1 * r1 * (particle.pbest_position - particle.position)
                + c2 * r2 * (gbest_position - particle.position)
            )
            particle.position = particle.position + particle.velocity
            if particle.position < lower_bound:
                particle.position = lower_bound
            if particle.position > upper_bound:
                particle.position = upper_bound

            print(f"iteration {iteration+1}: best workfroce = {gbest_position:.2f}, profit = {gbest_value:.2f}")

    return gbest_position, gbest_value

if __name__ == "__main__":
    best_pos, best_val = PSO()()
    print("\nfinal Result:")
    print(f"op worksize: {best_pos:.2f}, max profit: {best_val:.2f}")

```

Op:

```

iteration 1: best workfroce = 17.93, profit = 167.86
iteration 2: best workfroce = 18.31, profit = 168.56
iteration 3: best workfroce = 18.71, profit = 169.17
iteration 4: best workfroce = 19.89, profit = 169.99
iteration 5: best workfroce = 19.89, profit = 169.99
iteration 6: best workfroce = 20.03, profit = 170.00
iteration 7: best workfroce = 20.03, profit = 170.00
iteration 8: best workfroce = 20.03, profit = 170.00
iteration 9: best workfroce = 20.03, profit = 170.00
iteration 10: best workfroce = 19.99, profit = 170.00

final Result:
op worksize: 19.99, max profit: 170.00

```

## Program 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 route that visits a list of cities and returns to the origin city.

Algorithm:

LAB-5 Ant Colony Optimization

- We can use TSP using this algo
- Ant releases pheromone chemical on its way
- We consider pheromone and cost matrix to find out the best path.

→ PHEROMONE  
→ DECISION MAKING

cost matrix → gives length of the edge  
pheromone matrix → gives quantity of pheromolecules path

$$\Delta T_{ij}^k = \begin{cases} \frac{1}{L_k} & k^{th} \text{ ant travels on edge } i,j \\ 0 & \Delta T \rightarrow \text{say pheromone value} \end{cases}$$

It is inverse of length  
Length ↑ Pheromone ↓

\* As time goes pheromone gets evaporated

$$T_{ij} = \sum_{k=1}^m \Delta T_{ij}^k \quad (\text{without evaporation})$$

$$T_{ij}^k = (1 - \rho) T_{ij} + \sum_{k=1}^m \Delta T_{ij}^k \quad (\text{with evaporation})$$

for e.g.

cost graph

pheromone graph

$$T_{ij} = \sum_{k=1}^m \Delta T_{ij}^k$$

$$(l) L_1 = 14 \rightarrow \Delta T_{ij}^1 = \frac{1}{14}$$

$$(m) L_2 = 31 \rightarrow \Delta T_{ij}^2 = \frac{1}{31}$$

$$T_{ij} = (1 - \rho) T_{ij} + \sum_{k=1}^m \Delta T_{ij}^k$$

(where  $\rho$ )

$\rho$  takes value from 0 to 1  
if it is 1 → no evaporation

Now we need to calculate probability of choosing edge.

~~$$P_{ij} = \frac{(T_{ij})^\eta \cdot (T_{ij})^{\eta}}{\sum ((T_{ij})^\eta \cdot (T_{ij})^{\eta})}$$~~

$$P_{ij} = \frac{(T_{ij})^\eta \cdot (T_{ij})^{\eta}}{\sum ((T_{ij})^\eta \cdot (T_{ij})^{\eta})}, \eta \rightarrow \frac{1}{L_{ij}}$$

pheromone × 1  
length ] In short

We use roulette based sel.  
prob [ 0.76 | 0.19 | 0.05 ]

Cumulative sum [ 1 | 0.24 | 0.05 ]

EDGE

Scanned with CamScanner

### Applicatp. TSP

We must visit n cities exactly once and return to start city (Christofides' algorithm)

prob of choosing next city

- Pheromone trail
- Visibility  $C^*(dist)$

algo →

- 1)  $m \rightarrow$  no. ants
- $n \rightarrow$  no. cities

$$\eta_{ij} = \frac{1}{d_{ij}} \quad (\text{for edge})$$

2) from i to j city

Cant start from random city

$$P_{ij} = \frac{(\eta_{ij}^\alpha)(\eta_{ij}^P)}{\sum_{k \neq i} (\eta_{ik}^\alpha)(\eta_{ik}^P)}$$

$\alpha \rightarrow$  pheromone (ctrl)

$P \rightarrow$  heuristic (dist)

3) Evaporate Evaporate and Deposit

~~$T_{ij} \leftarrow T_{ij} - Q$~~

4) Update best path

5) Return

### pseudo code

$$T(i)[j] = 10$$

$$\eta[i][j] = 1 / dist[i][j]$$

best length = 0

best tour = None

For iteration  $k$  to max:

for each ant  $k=1$  to m:

start (random) city

while not visited ws:

next city  $j$  with prob.  $P(j|C_i)$

move to city  $j$

Total len  $L[k]$

$\Sigma_j L[k]$  < best\_length:

bestlength =  $L[k]$

best tour = current tour

for edges  $(i,j)$ :

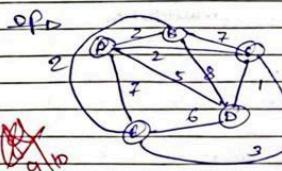
Evaporate phenomena:  $T(i)[j] = (1 - P)^a T(i)[j]$

For each ant  $k$ :

for each edge  $(i,j)$  in ant k's tour:

$$T(i)[j] += Q / L[k]$$

Return best tour, best length



Iteration: Best Path Length = 13

A → B → F → C → D → 0  
Total cost: 13



Scanned with CamScanner

Code:

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

```
class AntColonyTSP:
```

```
    def __init__(self, graph, n_ants, n_iterations, alpha, beta, evaporation_rate,
    pheromone_init, Q=1):
        self.graph = graph
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.Q = Q
        self.pheromone = np.ones_like(graph) * pheromone_init
        self.best_path = None
        self.best_path_length = float('inf')
```

```
def _calculate_transition_probabilities(self, ant, visited):
```

```
    current_node = ant[-1]
```

```
    probabilities = []

```

```
    for j in range(len(self.graph)):
```

```
        if j not in visited:
```

```
            pheromone = self.pheromone[current_node][j] ** self.alpha
```

```
            distance = (1.0 / self.graph[current_node][j]) ** self.beta
```

```
            probabilities.append(pheromone * distance)
```

```

        else:
            probabilities.append(0)
    total_pheromone = sum(probabilities)
    if total_pheromone == 0:
        return [0 for _ in probabilities]
    probabilities = [p / total_pheromone for p in probabilities]
    return probabilities
def __construct_path(self, start_node):
    visited = set([start_node])
    path = [start_node]
    total_distance = 0
    while len(path) < len(self.graph):
        current_node = path[-1]
        probabilities = self._calculate_transition_probabilities(path, visited)
        next_node = self._select_next_node(probabilities)
        visited.add(next_node)
        path.append(next_node)
        total_distance += self.graph[current_node][next_node]
    total_distance += self.graph[path[-1]][path[0]]
    path.append(path[0])
    return path, total_distance

def _select_next_node(self, probabilities):
    return np.random.choice(len(probabilities), p=probabilities)

def _update_pheromones(self, paths, path_lengths):
    self.pheromone *= (1 - self.evaporation_rate)
    for path, length in zip(paths, path_lengths):
        pheromone_deposit = self.Q / length
        for i in range(len(path) - 1):
            self.pheromone[path[i]][path[i + 1]] += pheromone_deposit
        self.pheromone[path[-1]][path[0]] += pheromone_deposit

def run(self, start_node):
    for iteration in range(self.n_iterations):
        paths = []
        path_lengths = []
        for _ in range(self.n_ants):
            path, length = self._construct_path(start_node)
            paths.append(path)
            path_lengths.append(length)
            if length < self.best_path_length:
                self.best_path_length = length
                self.best_path = path
        self._update_pheromones(paths, path_lengths)
        print(f"iteration {iteration + 1}: best path len = {self.best_path_length}")
    return self.best_path, self.best_path_length

```

```

graph = np.array([
    [0, 29, 20, 21, 16, 31, 100, 12],
    [29, 0, 15, 29, 28, 40, 72, 21],
    [20, 15, 0, 15, 14, 25, 81, 9],
    [21, 29, 15, 0, 4, 12, 92, 24],
    [16, 28, 14, 4, 0, 16, 94, 30],
    [31, 40, 25, 12, 16, 0, 95, 41],
    [100, 72, 81, 92, 94, 95, 0, 79],
    [12, 21, 9, 24, 30, 41, 79, 0]
])

```

```

n_ants = 30
n_iterations = 50
alpha = 1.2
beta = 3.0
evaporation_rate = 0.4
pheromone_init = 0.2
Q = 10

aco_tsp = AntColonyTSP(graph, n_ants, n_iterations, alpha, beta, evaporation_rate,
pheromone_init, Q)
best_path, best_path_length = aco_tsp.run(start_node=0)
print(f"\nbest path: {best_path}")
print(f"best path len: {best_path_length}")

```

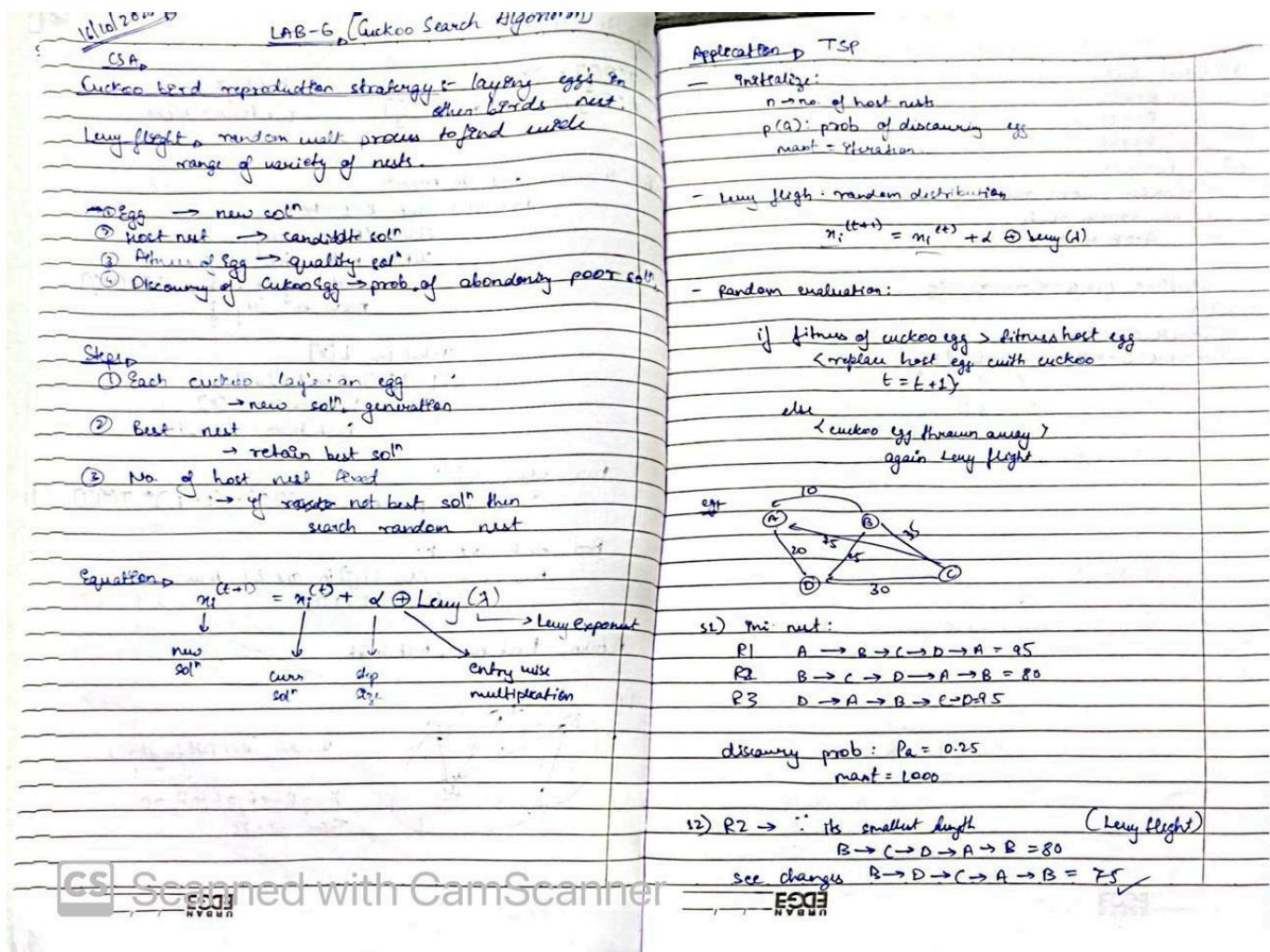
Op:

iteration 1: best path len = 247	iteration 28: best path len = 235
iteration 2: best path len = 247	iteration 29: best path len = 235
iteration 3: best path len = 235	iteration 30: best path len = 235
iteration 4: best path len = 235	iteration 31: best path len = 235
iteration 5: best path len = 235	iteration 32: best path len = 235
iteration 6: best path len = 235	iteration 33: best path len = 235
iteration 7: best path len = 235	iteration 34: best path len = 235
iteration 8: best path len = 235	iteration 35: best path len = 235
iteration 9: best path len = 235	iteration 36: best path len = 235
iteration 10: best path len = 235	iteration 37: best path len = 235
iteration 11: best path len = 235	iteration 38: best path len = 235
iteration 12: best path len = 235	iteration 39: best path len = 235
iteration 13: best path len = 235	iteration 40: best path len = 235
iteration 14: best path len = 235	iteration 41: best path len = 235
iteration 15: best path len = 235	iteration 42: best path len = 235
iteration 16: best path len = 235	iteration 43: best path len = 235
iteration 17: best path len = 235	iteration 44: best path len = 235
iteration 18: best path len = 235	iteration 45: best path len = 235
iteration 19: best path len = 235	iteration 46: best path len = 235
iteration 20: best path len = 235	iteration 47: best path len = 235
iteration 21: best path len = 235	iteration 48: best path len = 235
iteration 22: best path len = 235	iteration 49: best path len = 235
iteration 23: best path len = 235	iteration 50: best path len = 235
iteration 24: best path len = 235	
iteration 25: best path len = 235	best path: [0, 7, 2, 1, 6, 5, 3, 4, 0]
iteration 26: best path len = 235	best path len: 235
iteration 27: best path len = 235	

## Program 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.

**Algorithm:**



83) Select best .  
 R<sub>1</sub>=95  
 R<sub>2</sub>=75 ✓  
 R<sub>3</sub>=95  
 su) ∴ P<sub>a</sub>=0.25  
 abandon worst route  
 and new random route .  
 A → B → C → D → A  
 selectn → C → B → A → D → C = 70  
 (2) op  
 ∴ Iterate HU Best route :  
 Best route = B → A → D → C → B = 850  
 08  
 6/10

Code:

```

import random
import math

weights = [4, 8, 5, 3, 6, 7]
values = [12, 20, 16, 10, 18, 22]
capacity = 20
n_items = len(weights)
n_nests = 12
max_iter = 40
pa = 0.25

def fitness(solution):
    total_wt = sum(w for w, s in zip(weights, solution) if s == 1)
    total_val = sum(v for v, s in zip(values, solution) if s == 1)
    return total_val if total_wt <= capacity else 0

def repair(solution):
    while sum(w for w, s in zip(weights, solution) if s == 1) > capacity:
        ones = [i for i, s in enumerate(solution) if s == 1]
        if not ones:
            break
        solution[random.choice(ones)] = 0
    return solution

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)
    return u / (abs(v) ** (1 / Lambda))

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 repair(new_nest)

def cuckoo_search():
    nests = [generate_nest() for _ in range(n_nests)]
    fitness_vals = [fitness(repair(nest)) for nest in nests]
    best_idx = fitness_vals.index(max(fitness_vals))
    best_nest = nests[best_idx][:]
    best_fitness = fitness_vals[best_idx]
    for it in range(1, max_iter + 1):
        for i in range(n_nests):
            new_nest = get_cuckoo(nests[i], best_nest)
            new_fit = fitness(new_nest)
            if new_fit > fitness_vals[i]:
                nests[i], fitness_vals[i] = new_nest, new_fit
            num_abandon = int(pa * n_nests)
            worst_idx = sorted(range(len(fitness_vals)), key=lambda k:
fitness_vals[k])[:num_abandon]
            for idx in worst_idx:
                nests[idx] = generate_nest()
                fitness_vals[idx] = fitness(repair(nests[idx]))
            cur_best_idx = fitness_vals.index(max(fitness_vals))
            if fitness_vals[cur_best_idx] > best_fitness:
                best_fitness = fitness_vals[cur_best_idx]
                best_nest = nests[cur_best_idx][:]
        if it % 10 == 0:
            print(f"Iter {it}: Best val = {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("\n__final sol__")
    print(f"best nest: {best_solution}")
    print(f"total val: {best_value}")
    print(f"total weight: {total_weight}/{capacity}")

```

Op:

```

        Iter 10: Best val = 62
        Iter 20: Best val = 62
        Iter 30: Best val = 62
        Iter 40: Best val = 62

        __final sol__
        best nest: [1, 0, 0, 1, 1, 1]
        total val: 62
        total weight: 20/20

```

## 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.

Algorithm:

6/11/2023  
Grey Wolf Optimizer (GWO) LAB-7

- It's a nature inspired metaheuristic algorithm
- Mimics grey wolf's social hierarchy: Alpha, Beta, Delta, Omega.
- Hunting strategies: encircling, chasing & attacking prey

$\alpha \rightarrow$  leader  $\rightarrow$  Best fitness.  
 $\beta \rightarrow$  2nd best  $\rightarrow$  2nd "  
 $\gamma \rightarrow$  3rd best  $\rightarrow$  3rd "  
 $\omega \rightarrow$  follower  $\rightarrow$  4th "

### Formulas

D Distance to prey

$$D = |C \cdot x_p - x|$$

$x_p$   $\rightarrow$  pos<sup>n</sup> of prey (best sol<sup>n</sup>)

$x$   $\rightarrow$  curr. wolf pos<sup>n</sup>.

$C = 2 \cdot r_2, r_2 \in [0, 1]$  (randomization)

$r$  coeff. vector

2] Post<sup>n</sup>, update eq<sup>n</sup>

$$x(t+1) = x_p(t) - A \cdot D$$

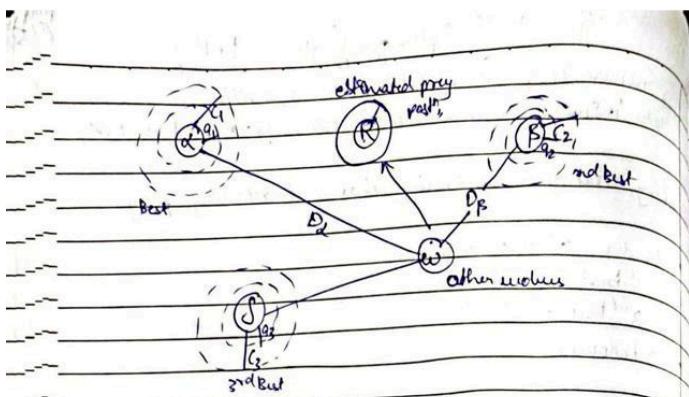
$A \rightarrow$  coeff. vector  $= 2 \cdot g \cdot r_1 - g$

$g \rightarrow$  vector that decreases linearly from 2 to 0 (decreasing iteration)

$r_1 \in [0, 1]$

### Algorithm

1. Init population, wolves
2. Evaluate fitness of each wolf
3. Identify  $\alpha, \beta, \gamma, \omega$
4. Update pos<sup>n</sup> of wolves
5. Handle boundaries
6. Repeat till last iteration
7. Return  $\alpha$ . wolf as best sol<sup>n</sup>.



### Applicatn → Feature selection

selecting most used / relevant feature from a large dataset  
to improve.

consider 1 = select feature

0 = discard feature

1. Init pos<sup>t</sup>  $X_i^t (i=1, 2, \dots, n)$  randomly

2. evaluate fitness  $f(x_i)$

3.  $\alpha, \beta, \delta$

4. For  $t=1$  to  $T$  do

~~for each wolf i do~~  $a = \text{random}$  situation

for each wolf  $i$  do

for each wolf  $j$  do

$x_1, x_2 = \text{random number } [0, 1]$

$A = 2 * a * x_1 - a$

$B = 2 * x_2$

$$D_\alpha = |C * X_\alpha(j) - X_1(j)| ; X_1 = X_\alpha(j) - D_\alpha$$

$$D_\beta = |C * X_\beta(j) - X_1(j)| ; X_2 = X_\beta(j) - D_\beta$$

$$D_\delta = |C * X_\delta(j) - X_1(j)| ; X_3 = X_\delta(j) - D_\delta$$

$$X_i(j) = \text{best}(x_1, x_2, x_3)$$

if  $X_i(j) > 0.5$  then  $\chi_i(j) = 1$  else  $\chi_i(j) = 0$

recalculate  $D(x_i)$

end for

Update  $\alpha, \beta, \delta$  based on new fitness

end for

return  $X_i$

~~88~~  
6/11

Opn Power demand

Total Power demand = 100

Total Precurv demand = 100

Surefian: 20 : Best cost (Alpha) = \$ 1895.2

" : 40 : " = \$ 1572.3

" : 60 : " = \$ 1527.6

" : 80 : " = \$ 1524.2

" : 100 : " = \$ 1524.5

CS Scanned with CamScanner

Code:

```
import numpy as np

def grey_wolf_optimizer(func, dimensions, limits, wolves=35, iterations=250):
    low, high = limits
    pack = np.random.uniform(low, high, (wolves, dimensions))
    alpha_pos, beta_pos, delta_pos = np.zeros(dimensions), np.zeros(dimensions),
    np.zeros(dimensions)
    alpha_fit = beta_fit = delta_fit = float("inf")
    track = np.zeros(iterations)

    for step in range(iterations):
        for j in range(wolves):
            pack[j] = np.clip(pack[j], low, high)
            score = func(pack[j])
```

```

    if score < alpha_fit:
        delta_fit, beta_fit = beta_fit, alpha_fit
        delta_pos, beta_pos = beta_pos.copy(), alpha_pos.copy()
        alpha_fit, alpha_pos = score, pack[j].copy()
    elif score < beta_fit:
        delta_fit, beta_fit = beta_fit, score
        delta_pos, beta_pos = beta_pos.copy(), pack[j].copy()
    elif score < delta_fit:
        delta_fit, delta_pos = score, pack[j].copy()

    a = 2 - (2 * step / iterations)
    for j in range(wolves):
        for leader, key in zip([alpha_pos, beta_pos, delta_pos], ["P", "Q", "R"]):
            r1, r2 = np.random.rand(dimensions), np.random.rand(dimensions)
            A = 2 * a * r1 - a
            C = 2 * r2
            D = np.abs(C * leader - pack[j])
            move = leader - A * D
            if key == "P":
                Xp = move
            elif key == "Q":
                Xq = move
            else:
                Xr = move
            pack[j] = (Xp + Xq + Xr) / 3

        track[step] = alpha_fit
        if (step + 1) % (iterations // 8) == 0:
            print(f"Step {step+1}/{iterations} → Current best = {alpha_fit:.5e}")

    return alpha_fit, alpha_pos, track

if __name__ == "__main__":
    def ackley(x):
        a, b, c = 20, 0.2, 2 * np.pi
        d = len(x)
        term1 = -a * np.exp(-b * np.sqrt(np.sum(x**2) / d))
        term2 = -np.exp(np.sum(np.cos(c * x)) / d)
        return term1 + term2 + a + np.e

dim = 50
limits = (-5, 5)
best_val, best_vector, progress = grey_wolf_optimizer(ackley, dim, limits, wolves=40, iterations=300)
print("\n__Final__")
print("\nOptimal Fitness:", best_val)
print("Top Wolf (first 10 dims):", best_vector[:10])

```

Op:

```
Step 37/300 → Current best = 2.48486e-01
Step 74/300 → Current best = 1.29794e-03
Step 111/300 → Current best = 2.13284e-05
Step 148/300 → Current best = 1.17326e-06
Step 185/300 → Current best = 1.51601e-07
Step 222/300 → Current best = 4.31687e-08
Step 259/300 → Current best = 1.95769e-08
Step 296/300 → Current best = 1.42710e-08
```

Final

```
Optimal Fitness: 1.4201941400671103e-08
Top Wolf (first 10 dims): [-4.65764821e-09 -3.15568744e-09 -3.28053611e-09 -2.63956540e-09
-4.26718578e-09 -3.49350096e-09 -3.63674779e-09 -3.77318141e-09
-4.07583716e-09 -3.67479989e-09]
```

## 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.

Algorithm:

**Parallel Cellular Algorithm**

Parallel cellular algo combines the parallel processing of algorithms with localised, grid-based structure of cellular automata.

**Key features:**

- Highly parallel & distributed
- Leverage the principles of cellular automata
- Suitable for large-scale optimization problems

**2 main ideas:**

- i) Biological Cells: Think of how cells in body work independently but interact with neighbours to create complex structures.
- ii) Cellular automata: Grid of cells, each in state ON/OFF that evolves with time.

**Algo →**

- 1) Initially create grid & randomly place cells in the grid.
- 2) Evaluate calc. the fitness of every cell.
- 3) Interaction / update
- 4) Iteration 2 to 3 no. of cycles until improvement
- 5) Termination

**Given →**

e.g. minimization  $f(n) = n^2 - 4n + 4$   
 $\therefore f'(n) = 2n - 4$   
 $n = 2$  with  $f(2) = 0$   
 $\therefore \text{Optimum value}$ .

**Exercise:**

grid cells: 

1	0	1
0	1	1
0	1	0

 $f(x) = 1 + f(n)$

a neighborhood: 

0	1	0	0
1	0	0	0
0	0	1	0
0	0	0	1

 Imagine  $0 \rightarrow \text{ON}$   $1 \rightarrow \text{OFF}$

**pseudo code (Image Processing: Noise Removal):**

- i) Load noisy img..
- ii) Set parameters: no. of iteration, neighborhood size, step, weight calc.
- iii) Init. cellular grid with the noisy img.

a) Iterations:

- i) Create new grid to store updated vals.
- ii) Get neighborhood for each (calculate)
- iii) Update curr pixel val.
- iv) Replace curr grid with newgrid.

b) Save denoised img.

```

img = load_img("noisy.jpg")
h, w = img.shape

for i in range(h):
    new_img = copy(img)
    for j in range(w):
        for g in range(w):
            nbs = np.array([a, b, c,
                           d, x, e,
                           f, g, h])
            for di in [-1, 0, 1] for dj in [-1, 0, 1]:
                if di == 0 and dj == 0:
                    continue
                nb_val = sum(nbs * new_img[i+di, j+dj])
                if nb_val <= 128:
                    new_img[i, j] = min(255, max(0, nb_val))
                else:
                    new_img[i, j] = max(0, min(255, nb_val))

img = new_img
    
```

Code:

```

import numpy as np

def f(x):
    return x**2 - 4*x + 4

GRID_SIZE = (10, 10)
MAX_ITER = 100
SEARCH_SPACE = (-10, 10)
LEARNING_RATE = 0.5

grid = np.random.uniform(SEARCH_SPACE[0], SEARCH_SPACE[1], GRID_SIZE)
    
```

```

def evaluate_fitness(grid):
    return f(grid)

def get_neighbors(grid, i, j):
    rows, cols = grid.shape
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            ni = (i + di) % rows
            nj = (j + dj) % cols
            neighbors.append(grid[ni, nj])
    return np.array(neighbors)

for iteration in range(MAX_ITER):
    fitness_grid = evaluate_fitness(grid)
    new_grid = np.copy(grid)

    for i in range(GRID_SIZE[0]):
        for j in range(GRID_SIZE[1]):
            neighbors = get_neighbors(grid, i, j)
            neighbor_fitness = f(neighbors)
            best_neighbor = neighbors[np.argmin(neighbor_fitness)]
            new_grid[i, j] = grid[i, j] + LEARNING_RATE * (best_neighbor - grid[i, j])

    grid = new_grid

    best_idx = np.unravel_index(np.argmin(f(grid)), grid.shape)
    best_x = grid[best_idx]
    best_fitness = f(best_x)

    if iteration % 10 == 0 or iteration == MAX_ITER - 1:
        print(f"Iteration {iteration+1}/{MAX_ITER} -> Best x: {best_x:.6f}, f(x): {best_fitness:.6f}")

print("\nBest solution found:")
print(f"x = {best_x:.6f}")
print(f"f(x) = {best_fitness:.6f}")

```

Op:

```

Iteration 1/100 -> Best x: 1.979375, f(x): 0.000425
Iteration 11/100 -> Best x: 1.999996, f(x): 0.000000
Iteration 21/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 31/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 41/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 51/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 61/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 71/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 81/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 91/100 -> Best x: 2.000000, f(x): 0.000000
Iteration 100/100 -> Best x: 2.000000, f(x): 0.000000

Best solution found:
x = 2.000000
f(x) = 0.000000

```

## **Program 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

Algorithm:

Code:

Op:

