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 inComputer 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

Github Link:
https://github.com/Tarun-619/BIS-lab-1WA23CS015

Tarun S Sunaddi 1WA23CS015 V-G

S.no	Date	Topic	Pg no. marks	Sign
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	
4	11/9/25	LAB-4 (PSO)	10/10	
5	9/10/25	LAB-5 (ACO)	10/10	
6	16/10/25	LAB-6 (CSA) →	10/10	
7	6/11/25	LAB-7 (GWO)	10/10	
8	13/11/25	LAB-8 (PCA)	10/10	

10/10

LAB 1 → BIO-INSPIRED SYSTEM →

1] Genetic Algorithm for optimization problems

→ A genetic algo (GA) is a technique where computers search for better solⁿ, by imitating the process of natural evolution. It begins with a group of random possible solⁿ, which are tested for their performance and the parents are then combined using crossovers to form new solutions and sometimes small mutations are added to maintain variety. By repeating this process many times, the overall quality of solⁿ improves step by step.

Applications:

- solving optimization tasks
- Machine learning technique
- Engineering & design problems
- Game strategies
- Bioformatics 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 & NN
- Clustering
- Job scheduling

4] Particle swarm optimizatⁿ,
- Neural network
- clustering
- feature selectⁿ,

5] Parallel Cellular organizatⁿ
- Image processing
- Finance
- Environment modeling

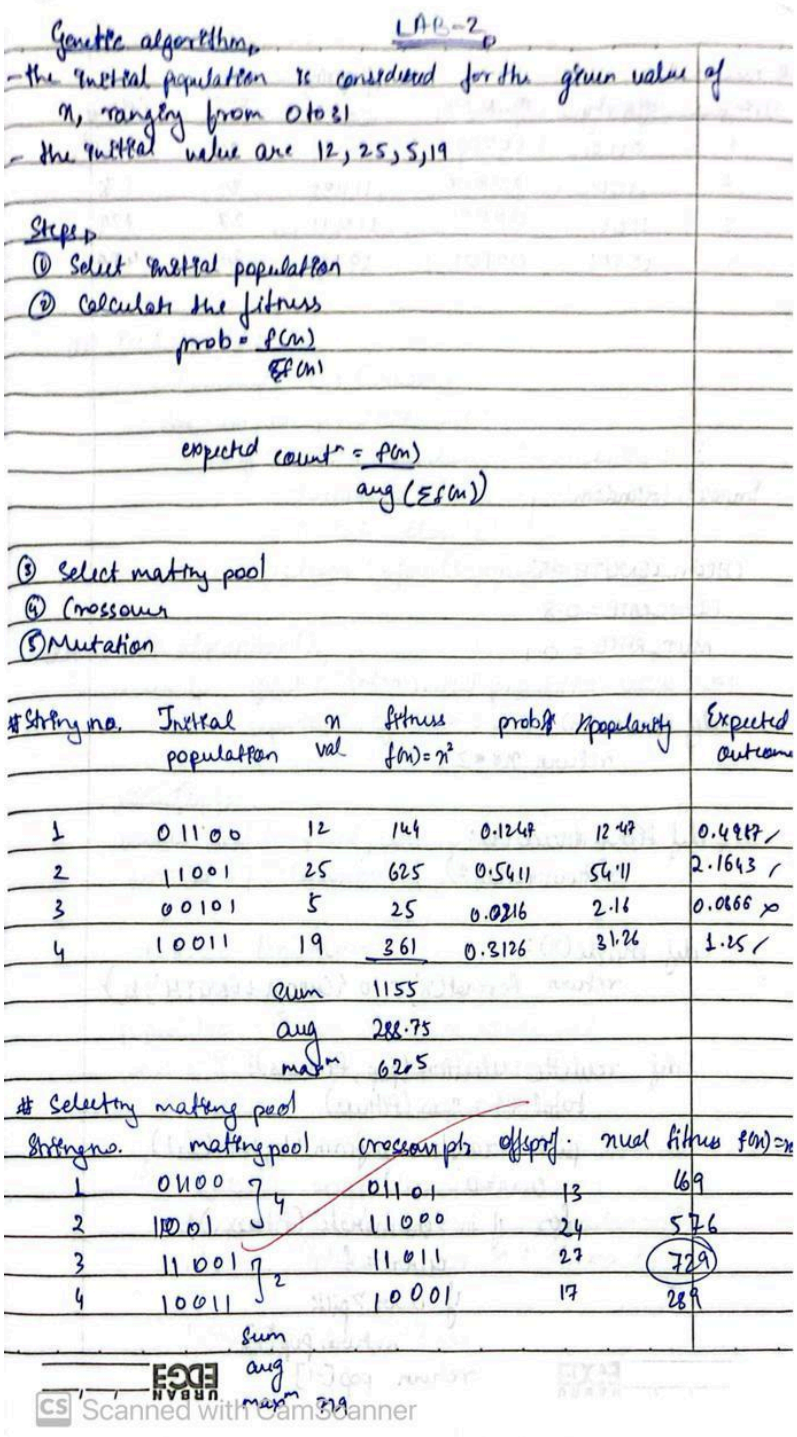
6] Optimization is a Gen exp, algo
- engineering design
- AI
- Finance

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:



Genetic algorithm — LAB-2

- the initial population is considered for the given value of x, ranging from 0 to 31
- the initial value are 12, 25, 5, 19

Steps:

① Select initial population
② Calculate the fitness
prob = f(x) / Σf(x)

expected count = f(x) / avg(Σf(x))

③ Select mating pool
④ Crossover
⑤ Mutation

# String no.	Initial population	x val	fitness f(x)=x²	prob#	%popularity	Expected outcome
1	01100	12	144	0.1248	12.48	0.4992 ✓
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	crossover pts	offspring	next fitness f(x)=x²
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289
		sum			
		avg			
		max	729		

mutation

Strings	offspring	Mutation	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

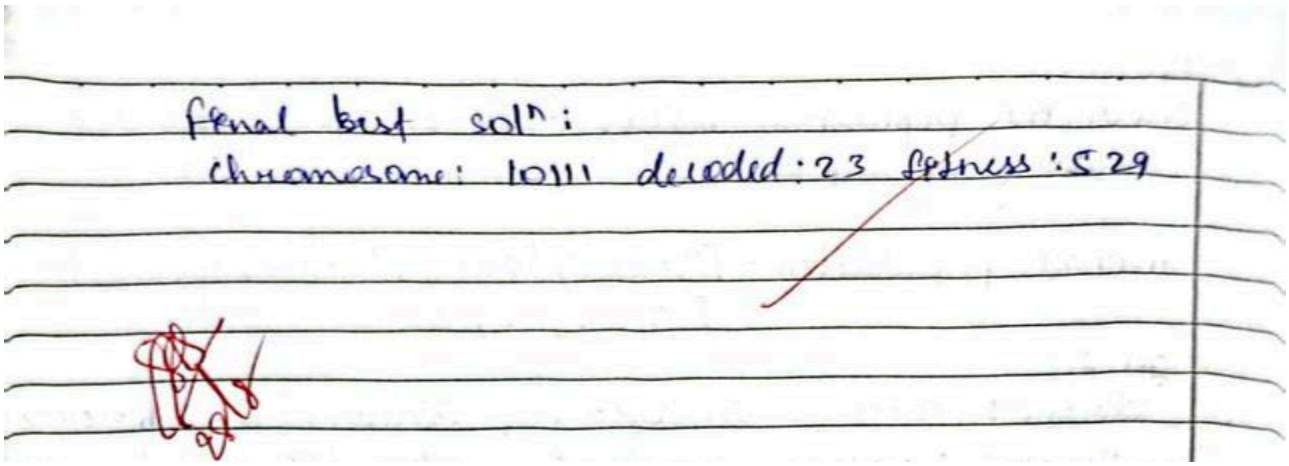
```
import random

CHROM_LENGTH = 5
CROSS_RATE = 0.8
MUT_RATE = 0.1

def fitness(x):
    return x**2

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

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

def roulette_selection(pop, fitness):
    total_fit = sum(fitness)
    pick = random.uniform(0, total_fit)
    curr = 0
    for i, f in enumerate(fitness):
        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 i in range(CHROM_LEN):
        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 64): ")
    values = list(map(int, user_ip.split()))
    gen = int(input("Enter no. of gen to run: "));

    POP_SIZE = len(values)

    population = [encode(x) for x in values]
    print("\n 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(x) for x in decoded]

        total_fit = sum(fitness)
```

```
    probs = [f/total_fit for f in fitness]
    expected = [p * possible 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 = {fitness[i]}, prob = {probs[i]:.3f},
        exp_count = {expected[i]:.2f}")

    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.extend([c1, c2])

    population = new_pop[:POP_SIZE]

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

genetic_algorithm()
```

output:

```
enter init population values (eg. 1 3 4 5): 3 6 7 2
num of generation: 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 = 0.31
chrom 3: 00111, decoded = 7, fitness = 49, prob = 0.50
chrom 4: 00010, decoded = 2, fitness = 4, prob = 0.04

gen 2:
chrom 1: 10011, decoded = 19, fitness = 361, prob = 0.364
chrom 2: 00011, decoded = 3, " = 9, " = 0.015
chrom 3: 00111, decoded = 7, " = 49, " = 0.08
chrom 4: 00101, decoded = 5, " = 25, " = 0.04
of

gen 3:
chrom 1: 10011, decoded = 19, fitness = 361, prob = 0.20
" 2: 10111, " = 23, " = 529, " = 0.29
" 3: 10111, " = 23, " = 529, " = 0.29
" 4: 10011, " = 19, " = 361, " = 0.20

gen 4:
chrom 1: 10011, decoded = 19, fitness = 361, prob = 0.23
" 2: 10111, " = 23, " = 529, " = 0.46
" 3: 10011, " = 19, " = 361, " = 0.23
" 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
```



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()
```

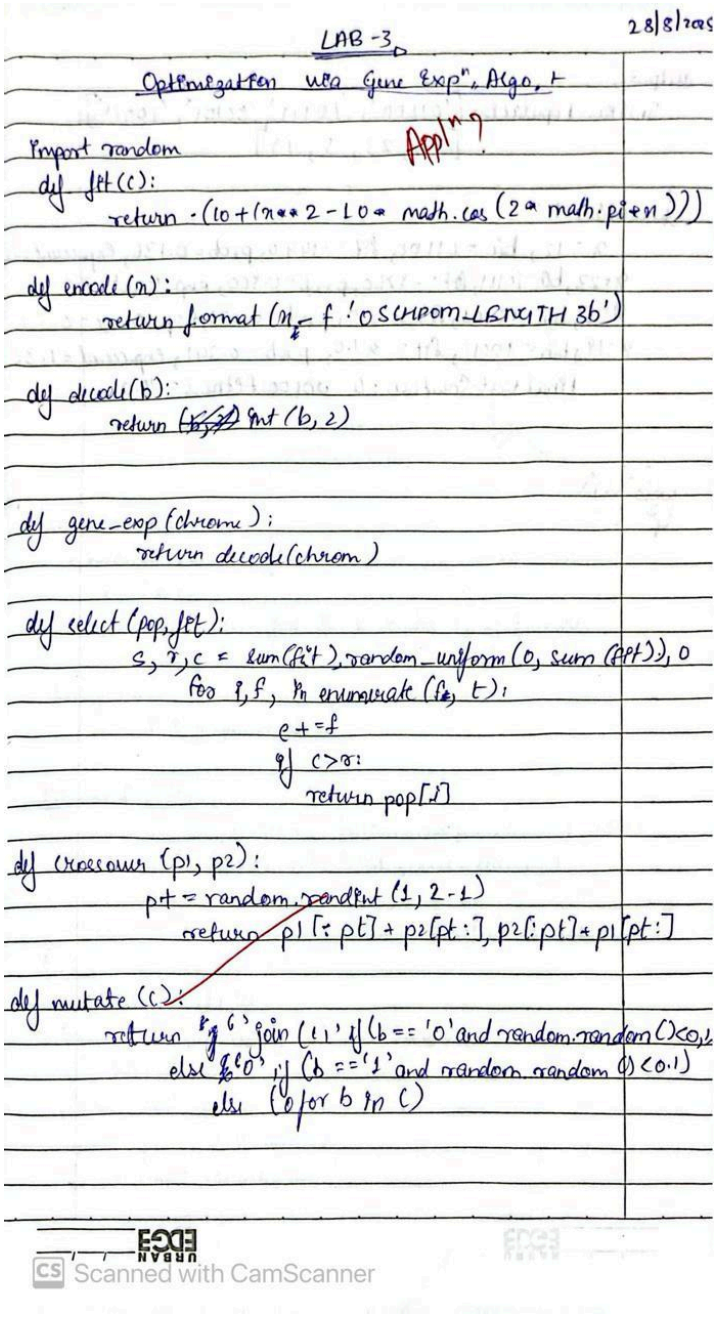
op:

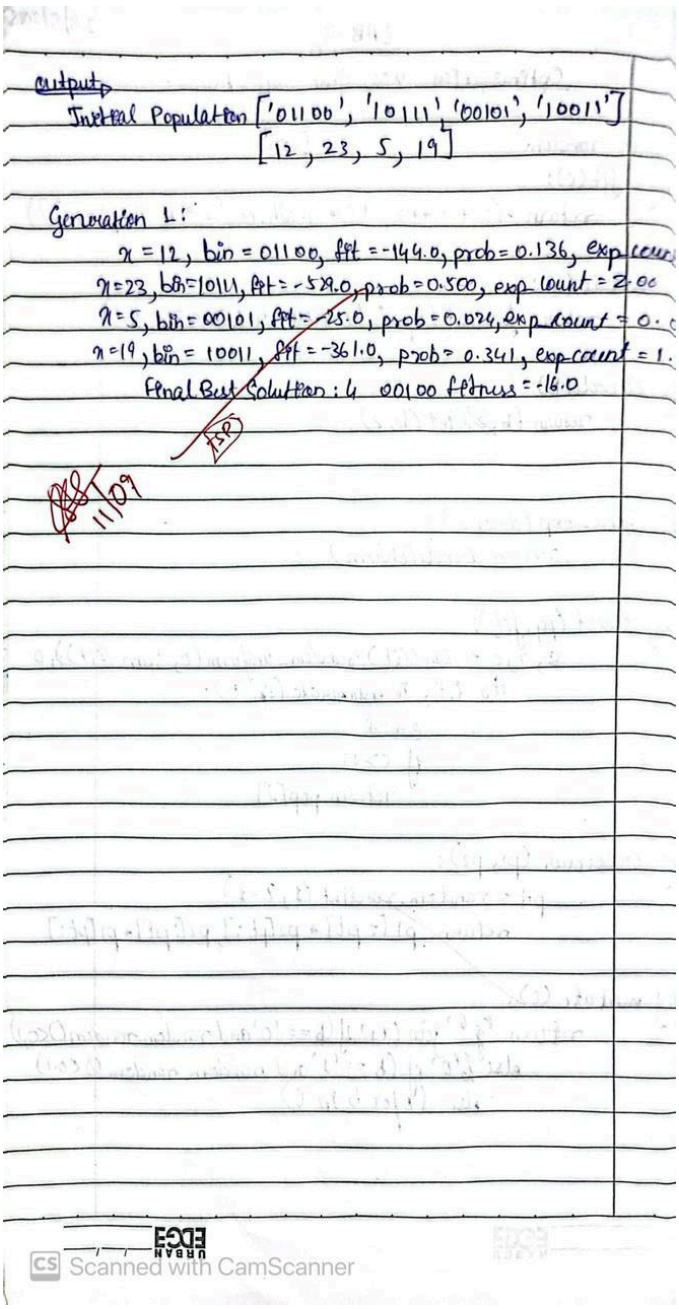
```
enter ini pop values (e.g. 1 2 3 4): 3 6 7 2
enter num of gen: 5
\ini pop: ['00011', '00110', '00111', '00010'] [3, 6, 7, 2]
\gen 1:
chromosome 1: 00011, decoded=3, fitness=9, prob=0.092
chromosome 2: 00110, decoded=6, fitness=36, prob=0.367
chromosome 3: 00111, decoded=7, fitness=49, prob=0.500
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:





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()
```

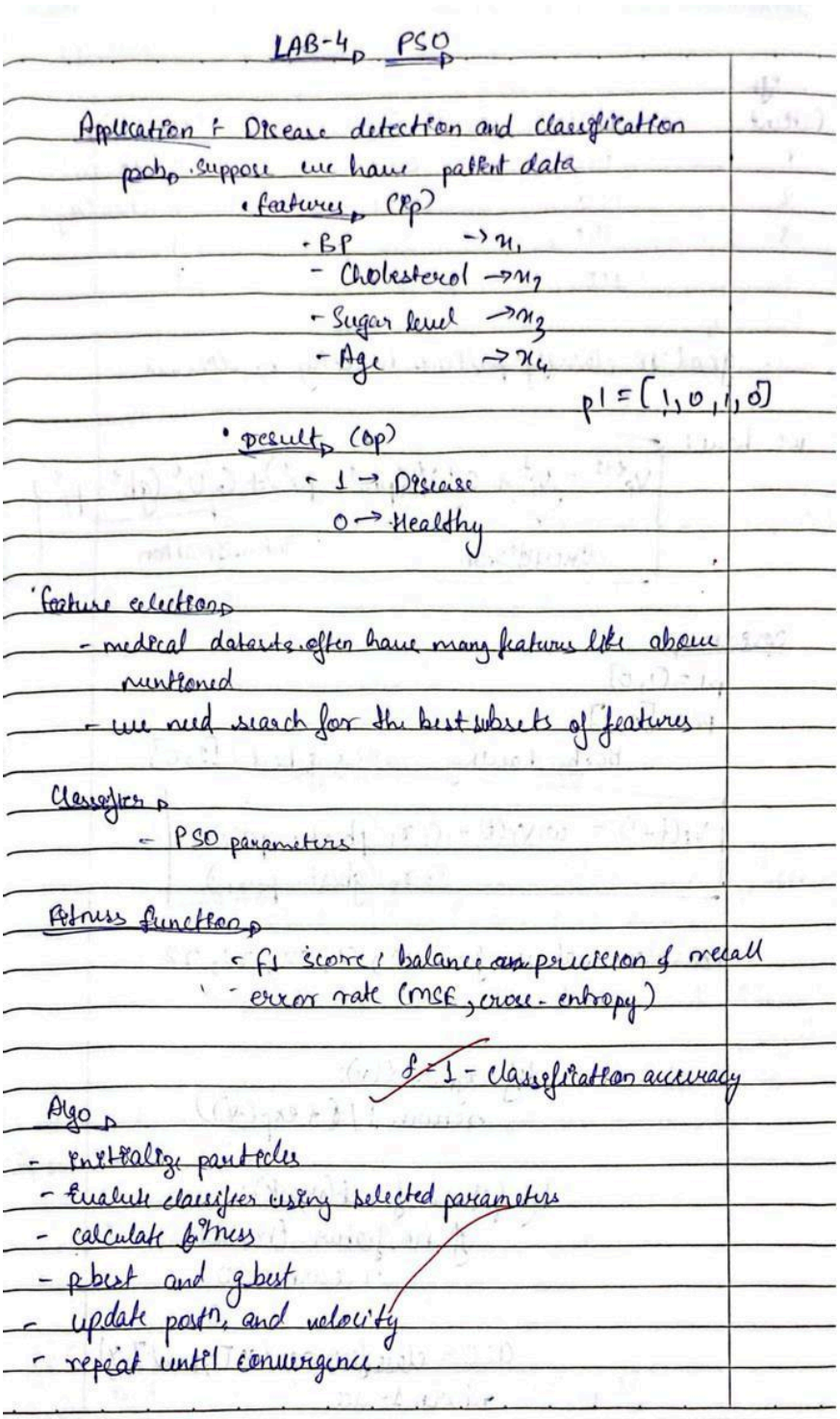
Op:

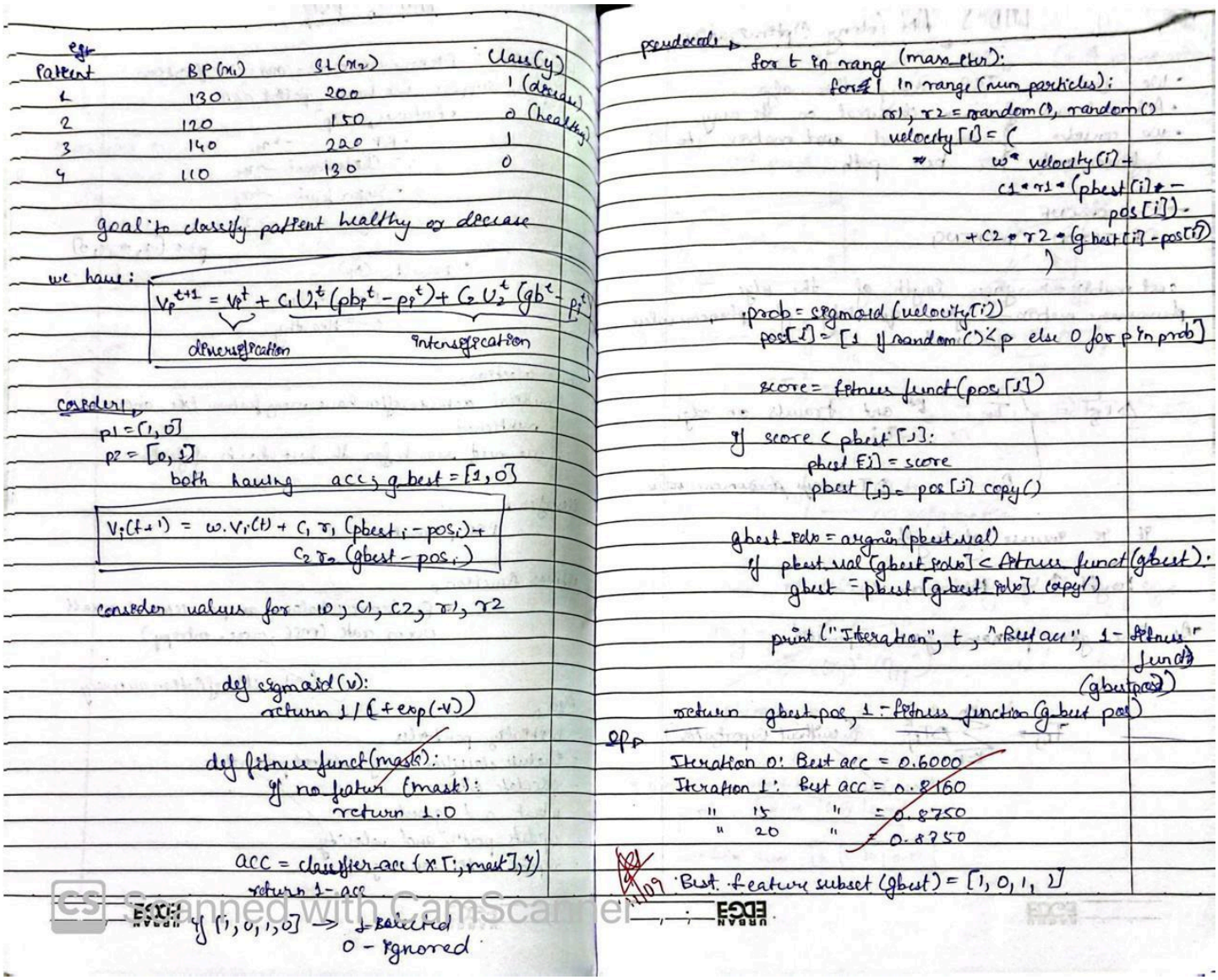
```
Initial Population: ['00111', '01111', '10110', '11100'] [7, 15, 22, 28]

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:





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}")
```

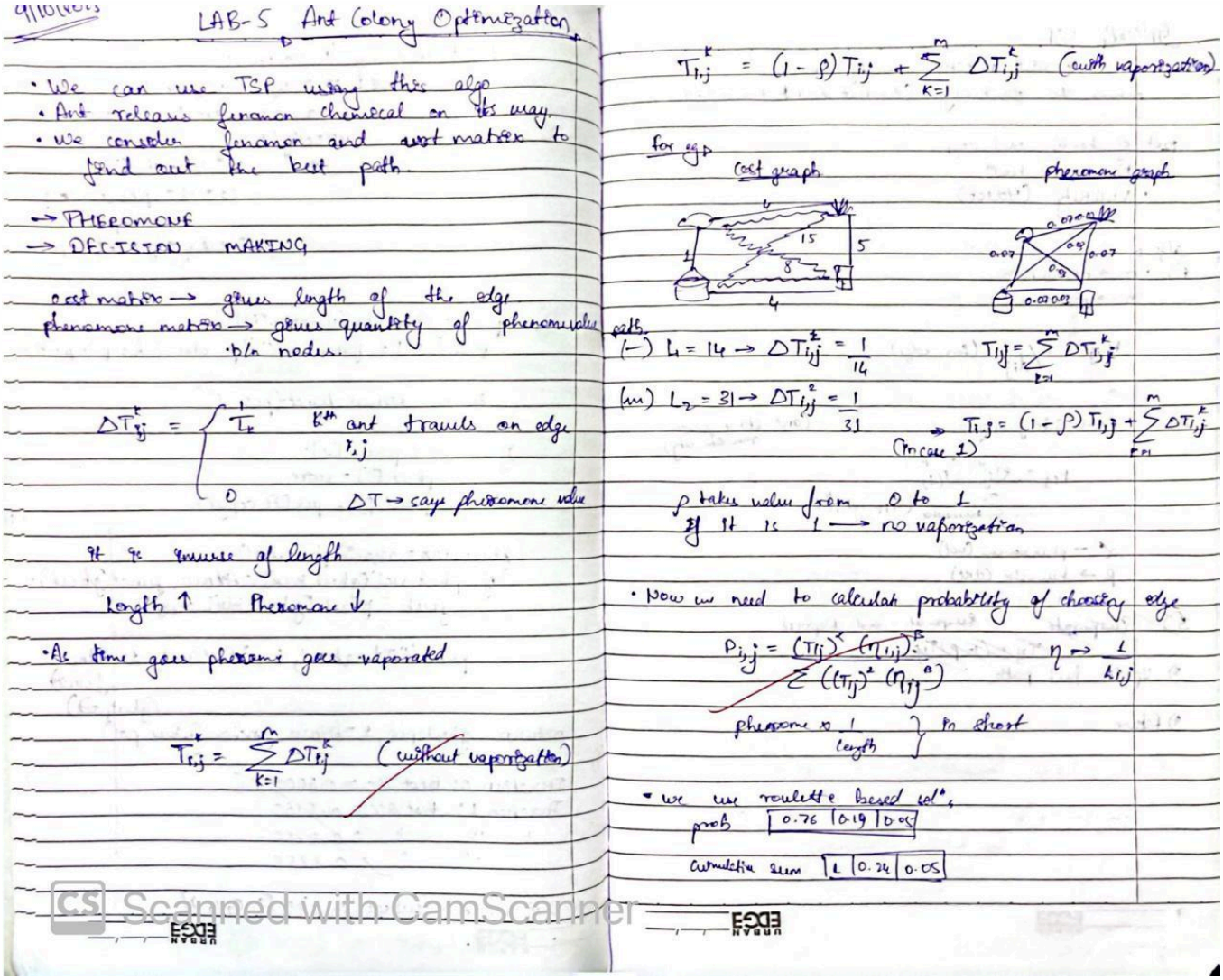
Op:

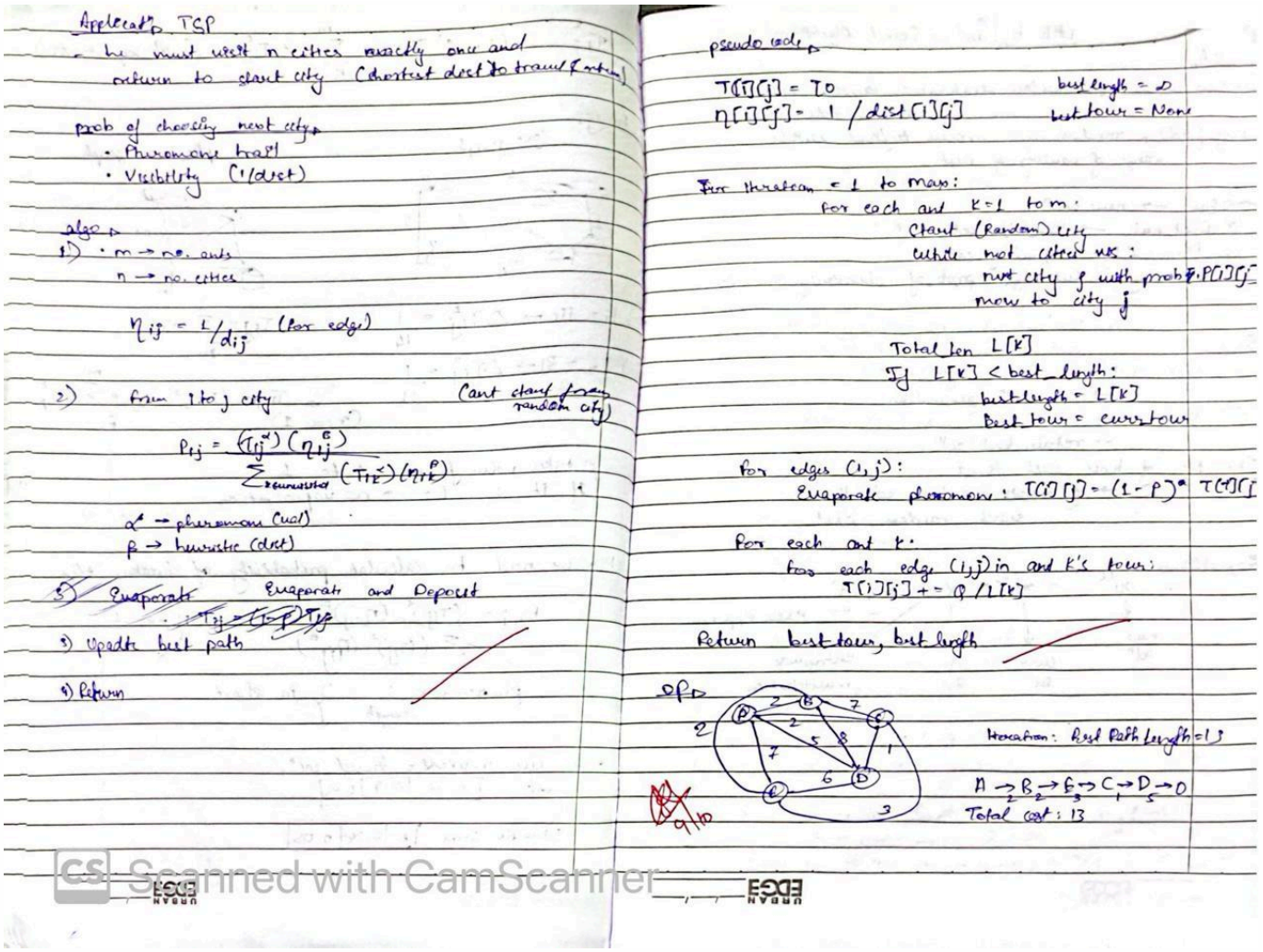
```
iteration 1: best workfroce = 17.93, profit = 167.86
iteration 2: best workfroce = 18.31, profit = 168.56
iteration 3: best workfroce = 18.71, profit = 169.17
iteration 4: best workfroce = 19.89, profit = 169.99
iteration 5: best workfroce = 19.89, profit = 169.99
iteration 6: best workfroce = 20.03, 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:





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}")
```

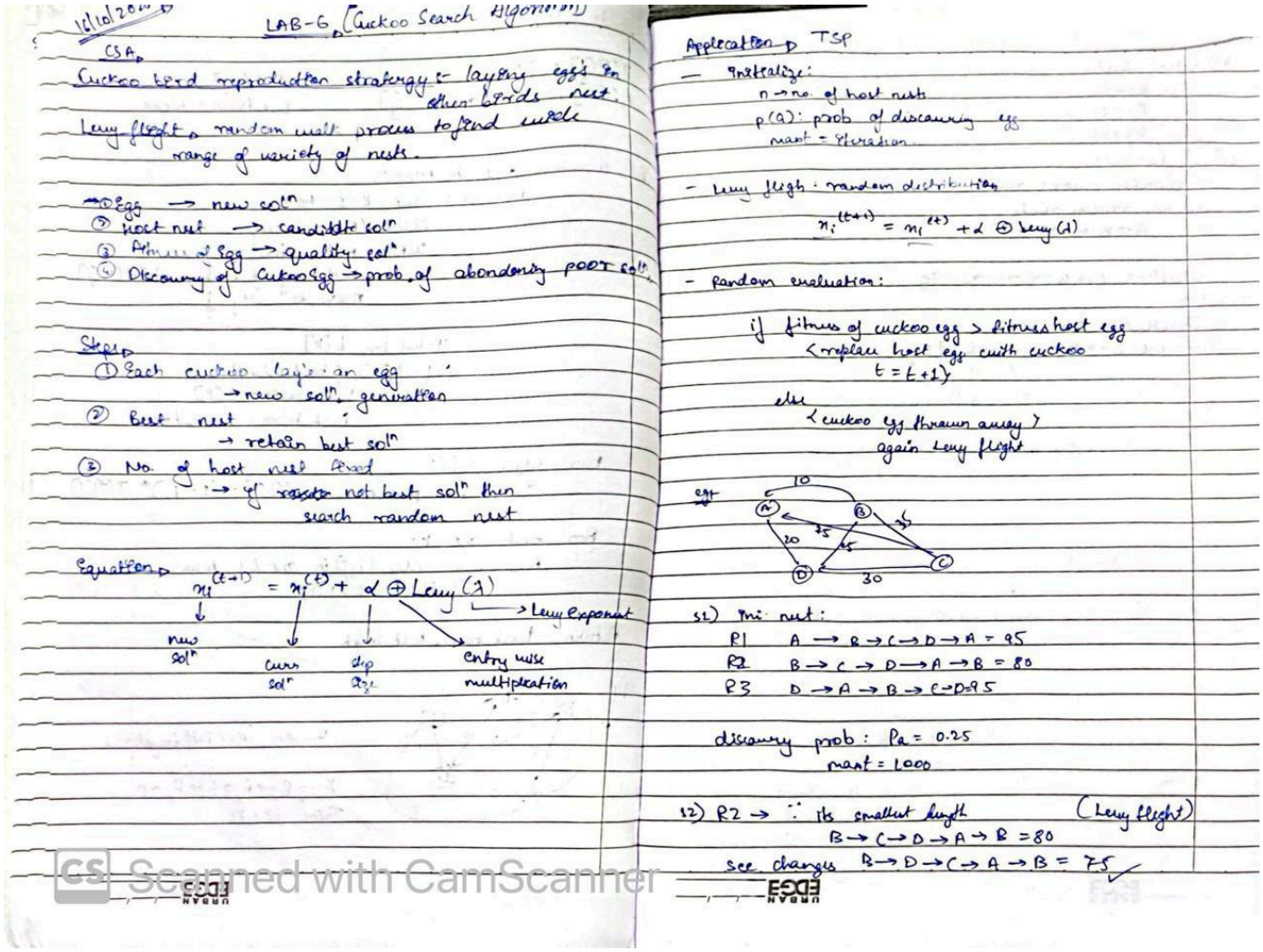
Op:

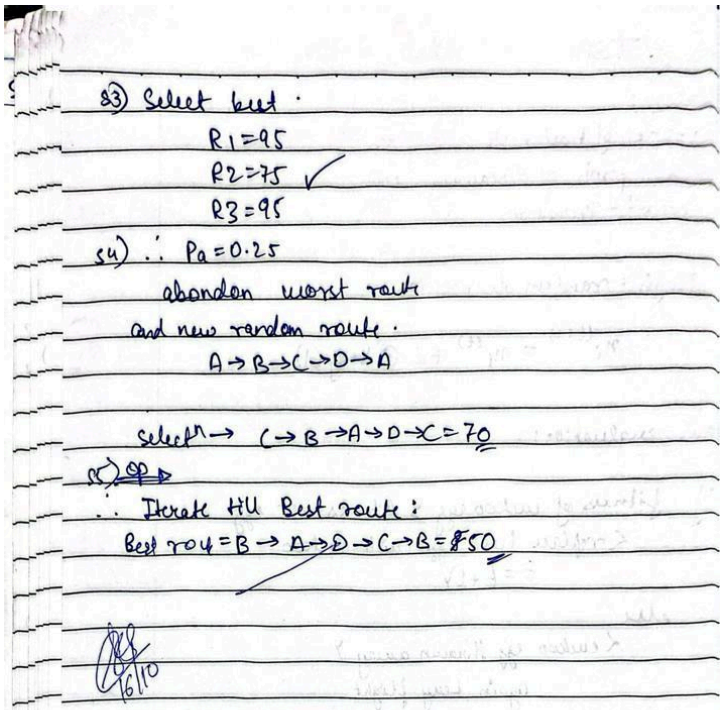
```
iteration 1: best path len = 247
iteration 2: best path len = 247
iteration 3: best path len = 235
iteration 4: best path len = 235
iteration 5: best path len = 235
iteration 6: best path len = 235
iteration 7: best path len = 235
iteration 8: best path len = 235
iteration 9: best path len = 235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

best path: [0, 7, 2, 1, 6, 5, 3, 4, 0]
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:





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}")
```

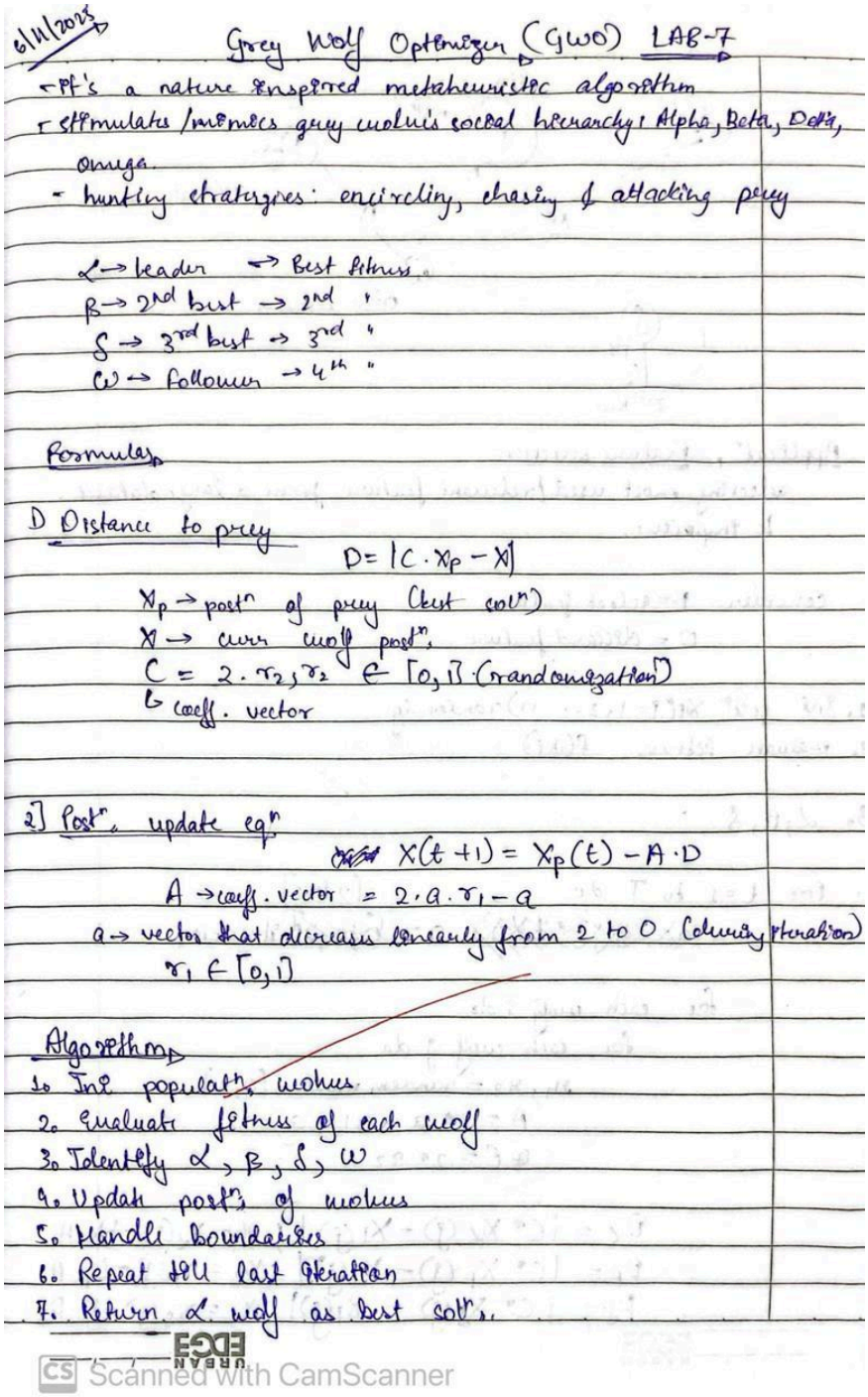
Op:

```
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
- stimulate / mimics grey wolf's social hierarchy: Alpha, Beta, Delta, Omega
- hunting strategies: encircling, chasing & attacking prey

α → leader → Best fitness
β → 2nd best → 2nd "
δ → 3rd best → 3rd "
ω → follower → 4th "

Formulas

1] Distance to prey

$$D = |C \cdot X_p - X|$$

X_p → post^n of prey (best soln)
X → curr wolf post^n
$C = 2 \cdot r_2$, $r_2 \in [0, 1]$ (randomization)
↳ coeff. vector

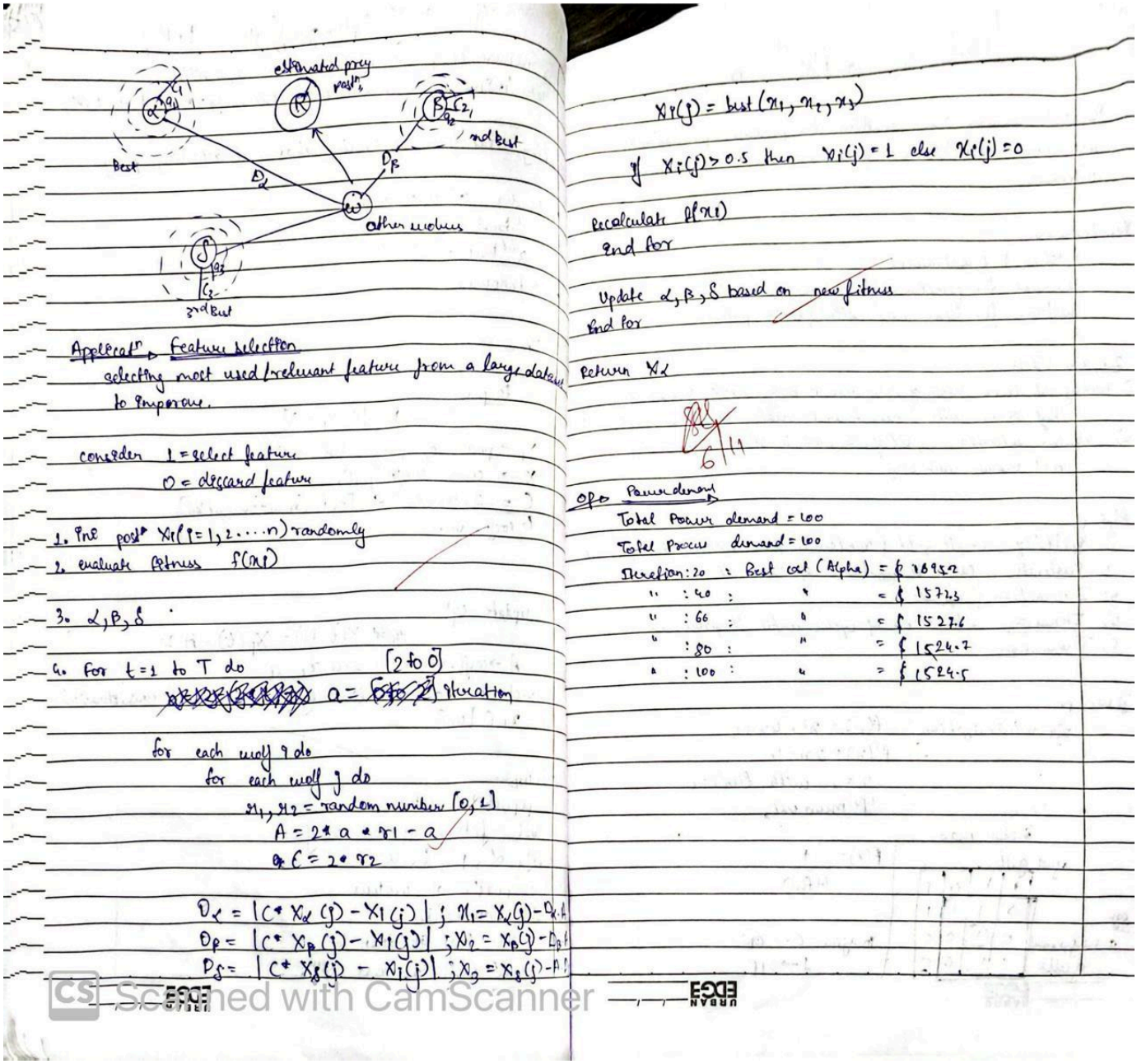
2] Post^n update eq^n

$$X(t+1) = X_p(t) - A \cdot D$$

A → coeff. vector $= 2 \cdot a \cdot r_1 - a$
a → vector that decreases linearly from 2 to 0 (during iteration)
$r_1 \in [0, 1]$

Algorithm

1. Init population wolves
2. Evaluate fitness of each wolf
3. Identify α, β, δ, ω
4. Update post^n of wolves
5. Handle boundaries
6. Repeat till last iteration
7. Return α wolf as best soln.



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])
```

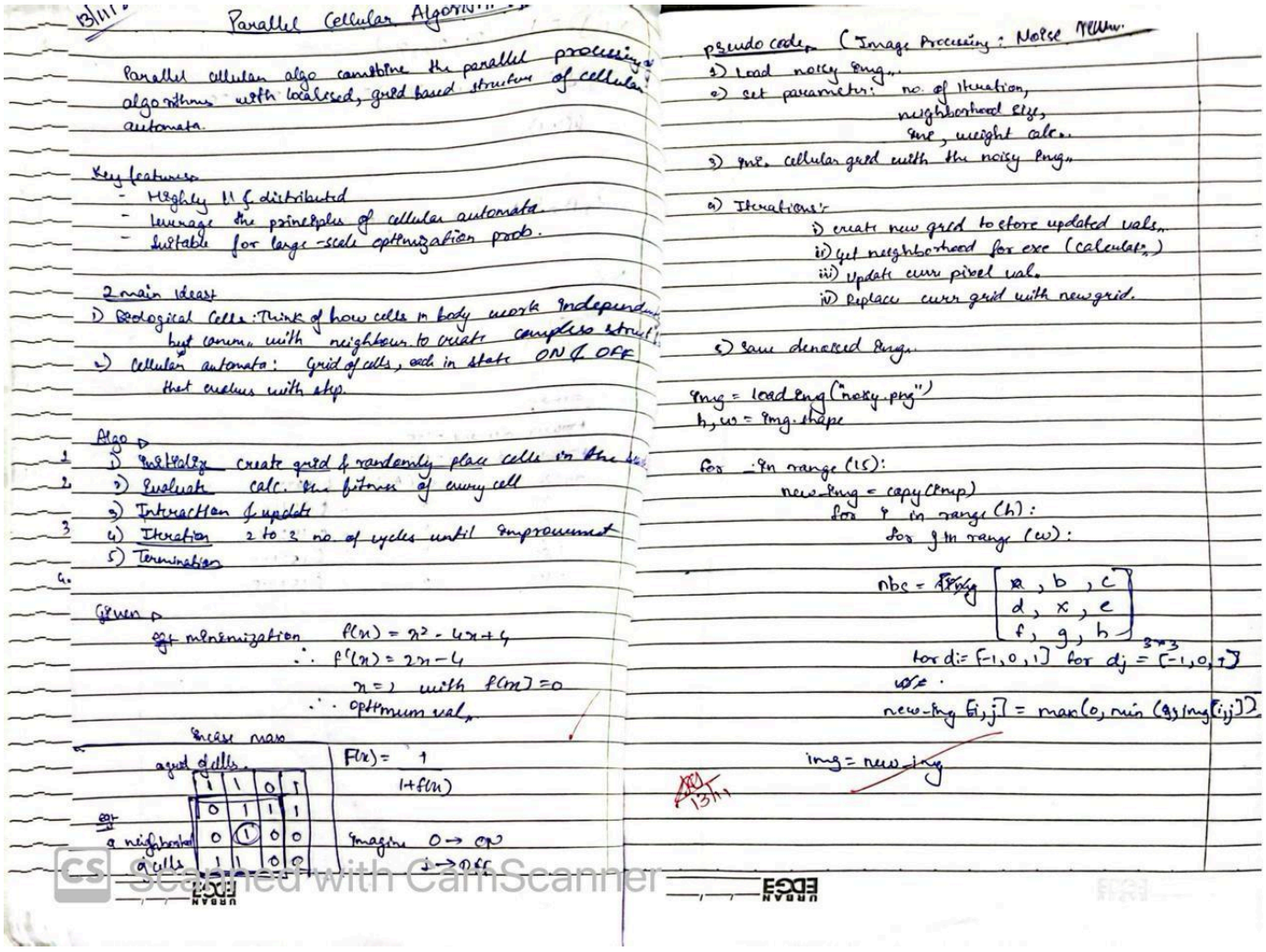
Op:

```
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 largescale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:



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: