VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shrinanda Shivprasad Dinde

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 Jan-2026

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 **Shrinanda Shivprasad Dinde(1BM23CS324),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

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

Index

Github Link:
https://github.com/shrinanda27/BIS-LAB.git

INDEX

Name Shrinanda Shivprasad Dinde

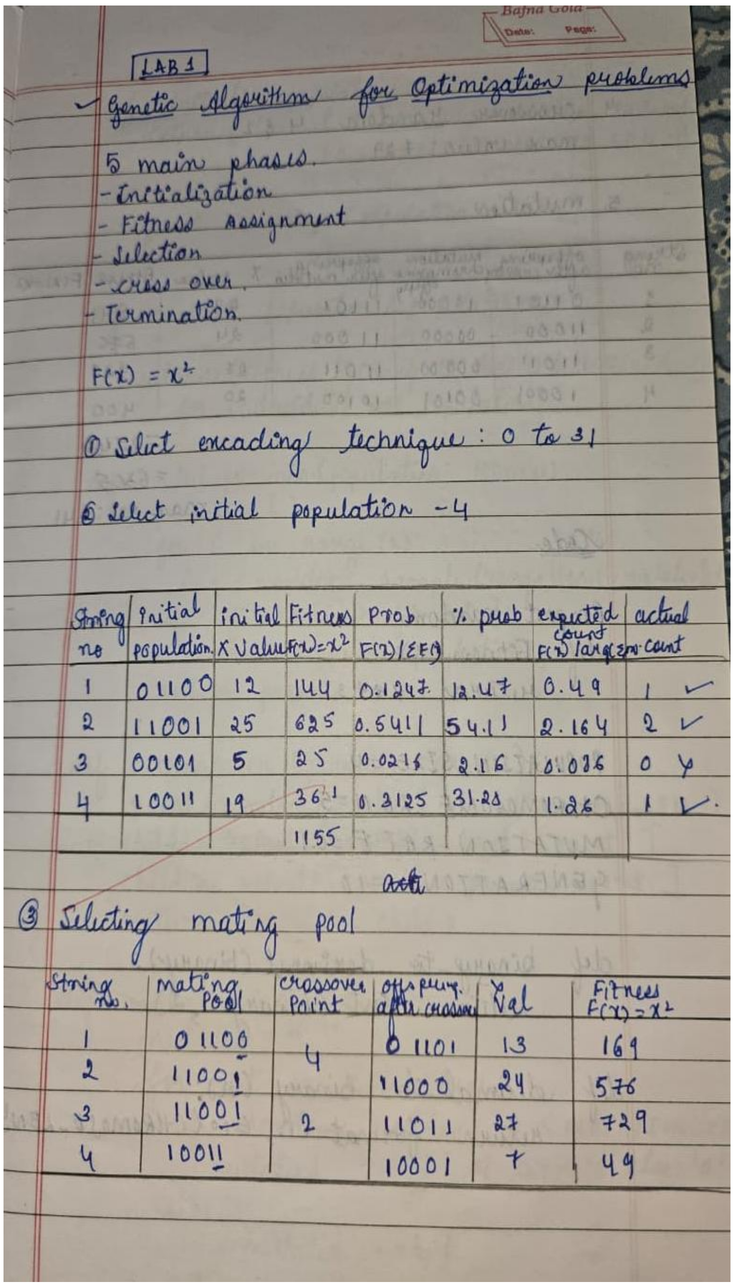
Standard Section 5F Roll No. 1BM23CS324

Subject BIS LAB

Program 1

Genetic Algorithm for Optimization Problems We have a set of jobs that must be completed and a limited amount of resources available to perform them. The challenge is to determine how to assign each job to the available resources in a way that minimizes total completion time, reduces overall cost, or maximizes efficiency. The goal is to find an optimal scheduling strategy under these constraints.

Observation:



LAB 1

Genetic Algorithm for Optimization problems

5 main phases.
- Initialization
- Fitness Assignment
- Selection
- cross over.
- Termination.

$F(x) = x^2$

① Select encoding technique : 0 to 31

② Select initial population - 4

String no	Initial population	initial X Value	Fitness $F(x)=x^2$	Prob $F(x)/\Sigma F(x)$	% prob	expected count $F(x)/avg$	actual count
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.086	0
4	10011	19	361	0.3125	31.25	1.25	1
			1155				

③ Selecting mating pool

String No.	mating pool	crossover Point	offspring after crossover	X Val	Fitness $F(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	7	49

Bafna Gold

Date: Page:

```
def initialize_population ():
    return [decimal_to_binary (random.radint
                      (0, 2** CHROMOSOME_LEN -1))

    for i in range (population_SIZE)]

def evaluate_population (population):

    return [Fitness (binary_to_decimal (individual))
        for individual in population]

def select_parents (population, Fitness):
    parents = []
    for i in range (2):
        i, j = random. sample (range(len(population
        if fitness[i] > fitness[j]:
            parents. append (population[j])
    return parents

def crossover (parent1, parent2):
    point = random. randient (1, CHROMOSOME_LEN-
    child1 = parent1[: point] + parent2[point :]
    child2 = parent2[: point] + parent1 [point :]
    return child 1, child 2

def mutate (individual):
    mutated = " "
    for bit in individual :
        if random . random () < MUTATION_
            mutated += '1' if bit=='0' else 'o
        else:
                mutated += bit
    return mutated
```

x1=x2

def genetic_algorithm()

OUTPUT:

Generation 1: Best = 28, Fitness = 784
Generation 2: Best = 28, Fitness = 784
Generation 3: Best = 29, Fitness = 841
Generation 4: Best = 29, Fitness = 841
Generation 5: Best = 30, Fitness = 900
Generation 6: Best = 31, Fitness = 961
Generation 7: Best = 31, Fitness = 961
Generation 8: Best = 30, Fitness = 900
Generation 9: Best = 30, Fitness = 900
Generation 10: Best = 30, Fitness = 900

Best Solution: 30
Fitness: 900

Code:

```
import random

def fitness(x):
    return x**2

def create_population(pop_size, lower_bound, upper_bound):
    return [random.randint(lower_bound, upper_bound) for _ in range(pop_size)]

def selection(population):
    tournament_size = 3
    selected = random.sample(population, tournament_size)
```

```
    selected = sorted(selected, key=fitness, reverse=True)
    return selected[0]

def to_binary_string(number, bits=32):
    if number < 0:
        return '-' + bin(abs(number))[2:].zfill(bits)
    else:
        return bin(number)[2:].zfill(bits)

def from_binary_string(binary_string):
    if binary_string.startswith('-'):
        return -int(binary_string[1:], 2)
    else:
        return int(binary_string, 2)

def crossover(parent1, parent2):
    b1 = to_binary_string(parent1)
    b2 = to_binary_string(parent2)
    cp = random.randint(1, len(b1.lstrip('-')) - 1)
    c1 = from_binary_string(b1[:cp] + b2[cp:])
    c2 = from_binary_string(b2[:cp] + b1[cp:])
    return c1, c2

def mutation(child, mutation_rate, lower_bound, upper_bound):
    if random.random() < mutation_rate:
        b = to_binary_string(child)
        mp = random.randint(1, len(b) - 1) if b.startswith('-') else
random.randint(0, len(b) - 1)
        bl = list(b)
        bl[mp] = '1' if bl[mp] == '0' else '0'
        child = from_binary_string(''.join(bl))
    return max(lower_bound, min(child, upper_bound))

def genetic_algorithm(pop_size, generations, mutation_rate, lower_bound,
upper_bound):
    population = create_population(pop_size, lower_bound, upper_bound)
    for g in range(generations):
        new_population = []
        for _ in range(pop_size // 2):
            p1 = selection(population)
            p2 = selection(population)
            c1, c2 = crossover(p1, p2)
            c1 = mutation(c1, mutation_rate, lower_bound, upper_bound)
            c2 = mutation(c2, mutation_rate, lower_bound, upper_bound)
            new_population.extend([c1, c2])
        population = new_population
```

```
        best = max(population, key=fitness)
        print(f"Generation {g+1}: Best solution = {best}, Fitness =
{fitness(best)}")
    return max(population, key=fitness)

pop_size = 5
generations = 4
mutation_rate = 0.01
lower_bound = 0
upper_bound = 31

best_solution = genetic_algorithm(pop_size, generations, mutation_rate, lower_bound,
upper_bound)
print(f"\nBest solution found: {best_solution}, Fitness = {fitness(best_solution)}")
```

Output:

```
••• Generation 1: Best solution = 22, Fitness = 484
    Generation 2: Best solution = 22, Fitness = 484
    Generation 3: Best solution = 22, Fitness = 484
    Generation 4: Best solution = 22, Fitness = 484

    Best solution found: 22, Fitness = 484
```

Program 2

Particle Swarm Optimization for Function Optimization Portfolio Optimization (Selecting assets) using Particle Swarm Optimization is about choosing how much money to allocate to different assets (stocks, bonds, etc.) to maximize expected return while minimizing risk (variance).

Algorithm:

LAB - 2

Particle Swarm Optimization

Pseudocode :-

1. Particle Initialization()
2. For i = 1 to max
   3. for each particle p int P do
      $f_p = f(P)$
4. If fp is better than f(pbest)
   pbest = p ;
5. end
6. end
7. gbest = best p in P
8. for each particle p in P do

9. $V_i^{t+1} = \underbrace{V_i^t}_{\text{inertia}} + \underbrace{C_1 U_1^t (pb_i^t - P_i^t)}_{\text{Personal influence}} + \underbrace{C_2 U_2^t (gb^t - P_i^t)}_{\text{social influence}}$

10. $P_i^{t+1} = P_i^t + V_i^{t+1}$
11. end
12. end

Bafna Gold
Date: Page:

Application problem

Rastrigin Function.

$$f(x) = A \cdot n + \sum_{i=1}^{n} [x_i^2 - A \cdot \cos(2\pi x_i)]$$

$A = 10$

n is dimensionality of search space

PSO Output

Best x: [-0.001982, 0.002997, 0.990737, -0.004
0.004635]

Best f(x): 1.0100547917422347

Iterations: 200, Particles: 50

First 5 history values:
[57.773033, 57.778033, 44.799763, 34.955526,
20.492025]

Last 5 history values:
[1.05793, 1.040874, 1.040874, 1.010055, 1.010055]

Iteration	Particle	x	f(x)	pbest pos	pbest fitness	gbest Pos	gbest fit
0	1	4.0	16.0	4.0	16.0		
0	2	-3.0	9.0	-3.0	9.0	-3.0	9.0
1	1	-2.3	5.29	-2.3	5.29		
1	2	-3.0	9.0	-3.0	9.0	-2.3	5.29

Code:

```python
import numpy as np


def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

def PSO(num_particles=30, dim=5, max_iter=200):
    w = 0.7
    c1 = 1.5
    c2 = 1.5
```

```python
    X = np.random.uniform(-5.12, 5.12, (num_particles, dim))
    V = np.zeros((num_particles, dim))

    pbest = X.copy()
    pbest_val = np.array([rastrigin(x) for x in X])

    gbest = pbest[np.argmin(pbest_val)]
    gbest_val = min(pbest_val)

    history = []


    for t in range(max_iter):
        for i in range(num_particles):

            r1, r2 = np.random.rand(), np.random.rand()
            V[i] = (
                w * V[i]
                + c1 * r1 * (pbest[i] - X[i])
                + c2 * r2 * (gbest - X[i])
            )

            X[i] = X[i] + V[i]


            X[i] = np.clip(X[i], -5.12, 5.12)


            f = rastrigin(X[i])


            if f < pbest_val[i]:
                pbest[i] = X[i]
                pbest_val[i] = f


        if min(pbest_val) < gbest_val:
            gbest = pbest[np.argmin(pbest_val)]
            gbest_val = min(pbest_val)

        history.append(gbest_val)

    return gbest, gbest_val, history
best_x, best_fx, history = PSO()

print("Best x:", best_x)
```

```python
print("Best f(x):", best_fx)
print("First 5 history values:", history[:5])
print("Last 5 history values:", history[-5:])
```

Output:

```
Best x: [ 2.18029186e-11  9.94958637e-01  1.98991223e+00 -9.94958640e-01
 -2.79413887e-09]
Best f(x): 5.969749304740667
First 5 history values: [np.float64(46.87644687644508), np.float64(46.87644687644508), np.float64(43.703638112789406), np.float64(34.13717413863966), np.float64(9.025079266891787)]
Last 5 history values: [np.float64(5.969749304740667), np.float64(5.969749304740667), np.float64(5.969749304740667), np.float64(5.969749304740667), np.float64(5.969749304740667)]
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem Ant Colony Optimization (ACO) for the Vehicle Routing Problem (VRP): It involves finding optimal routes for multiple vehicles to deliver goods to a set of customers from a central depot.

Algorithm:

Ant Colony Optimization (ACO) for the Traveling Salesman Problem

Algorithm

Input :-
A set of cities with known coordinates
Parameters: number of ants, α, β, rho, no. of iterations, initial pheromone.

Output :-
shortest possible route visiting all cities once and returning to start

Step 1: Initialize parameters
Calculate distance matrix.

Step 2: Repeat each iteration (until stopping criteria)

$$P_{ij} = \frac{(\text{pheromone}_{ij})^{\alpha} \times \left(\frac{1}{\text{dist}_{ij}}\right)^{\beta}}{\sum_{k \in \text{unvisited}} (\text{pheromone}_{ik})^{\alpha} \times \left(\frac{1}{\text{dist}_{ik}}\right)^{\beta}}$$

Step 3: Compare all ants routes
update global best route

Step 4: After reaching max number of iterations output best route found and its length

Example
Initialize

city	coordinates (x,y)
0	(1, 1)
1	(4, 1)
2	(4, 5)
3	(1, 5)

Distance matrix

From \ To	0	1	2	3
0	0	3.0	5.0	4.0
1	3.0	0	4.0	5.0
2	5.0	4.0	0	3.0
3	4.0	5.0	3.0	0

Initial pheromone matrix

From/To	0	1	2	3
0	0.1	0.1	0.1	0.1
1	0.1	0.1	0.1	0.1
2	0.1	0.1	0.1	0.1
3	0.1	0.1	0.1	0.1

Sol

Iteration	Best Route	Length	Example Pheromone on edge (0-1)	Example Pheromone on edge (0-3)
1	0→1→2→3→0	14	0.1214	0.05
2	0→3→2→1→0	14	0.0607	0.1321
3	1→0→3→2→1	14	~0.09	~0.1

MG
10/10/25.

Step 5:

$x = 0.5 \rightarrow f(x) = 0.25$

$x = 0 \quad f(x) = 0$

Global minimum found!

Eg

① $f(x) = x_1^2 + x_2^2$

Number of nests = 25

discovery rate = 0.25

100 iterations

25 nests in range [-5, 5]

Problem: Welded-Beam Design

minimize fabrication cost (material + welding) subject to constraints (stress, buckling, deflection, geometry).

x_1 = h (weld thickness) (in)

x_2 = l (weld length) (inches)

x_3 = t = beam height

x_4 = b = beam width

Minimize $f(x) = 1.10471\, x_1^2 x_2 + 0.04811\, x_3 x_4 (14 + x_2)$

① Initialize.

Set parameters n, P_a, T, λ, M.

penalized fitness

$$F(x) = f(x) + M \cdot \sum_i \max(0, g_i(x))^2$$

record x_{best}.

② loop $t=1$ to T

Generate new solutions by Levy flights.

③ New candidate: $y = x_i + \alpha \cdot S \odot (x_i - x_{best})$

Step scale

elementwise multiplic

④ Evaluate candidates

$F(y) < F(x_e)$

select worst $k = [p_a \cdot n]$ nests

⑤ update best.

⑥ ~~Adaptive~~ stopping check

stop when $t \geq T$

M.G. 17/10/25

Code:

```
import numpy as np


cities = np.array([
    [1, 1],
    [4, 1],
    [4, 5],
    [1, 5]
])


def distance_matrix(coords):
    n = len(coords)
    d = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            d[i][j] = np.linalg.norm(coords[i] - coords[j])
    return d
```

```python
dist = distance_matrix(cities)

alpha = 1
beta = 2
rho = 0.5
num_ants = 5
iterations = 3


pher = np.ones_like(dist) * 0.1

def probability(i, visited):
    probs = []
    for j in range(len(dist)):
        if j not in visited:
            tau = pher[i][j] ** alpha
            eta = (1 / dist[i][j]) ** beta
            probs.append((j, tau * eta))
    total = sum(p for _, p in probs)
    return [(node, p / total) for node, p in probs]

def choose_next(probs):
    r = np.random.random()
    cum = 0
    for node, p in probs:
        cum += p
        if r <= cum:
            return node
    return probs[-1][0]

def route_length(route):
    length = 0
    for i in range(len(route)):
        length += dist[route[i]][route[(i + 1) % len(route)]]
    return length

best_route = None
best_len = float("inf")

for it in range(iterations):
    all_routes = []

    for k in range(num_ants):
        start = 0
        route = [start]
```

```
        while len(route) < len(dist):
            probs = probability(route[-1], route)
            nxt = choose_next(probs)
            route.append(nxt)

        all_routes.append(route)


    pher = (1 - rho) * pher
    for r in all_routes:
        L = route_length(r)
        if L < best_len:
            best_len = L
            best_route = r
        for i in range(len(r)):
            a = r[i]
            b = r[(i + 1) % len(r)]
            pher[a][b] += 1 / L

    print(f"Iteration {it+1}: Best Route = {best_route}, Length = {best_len}")
```

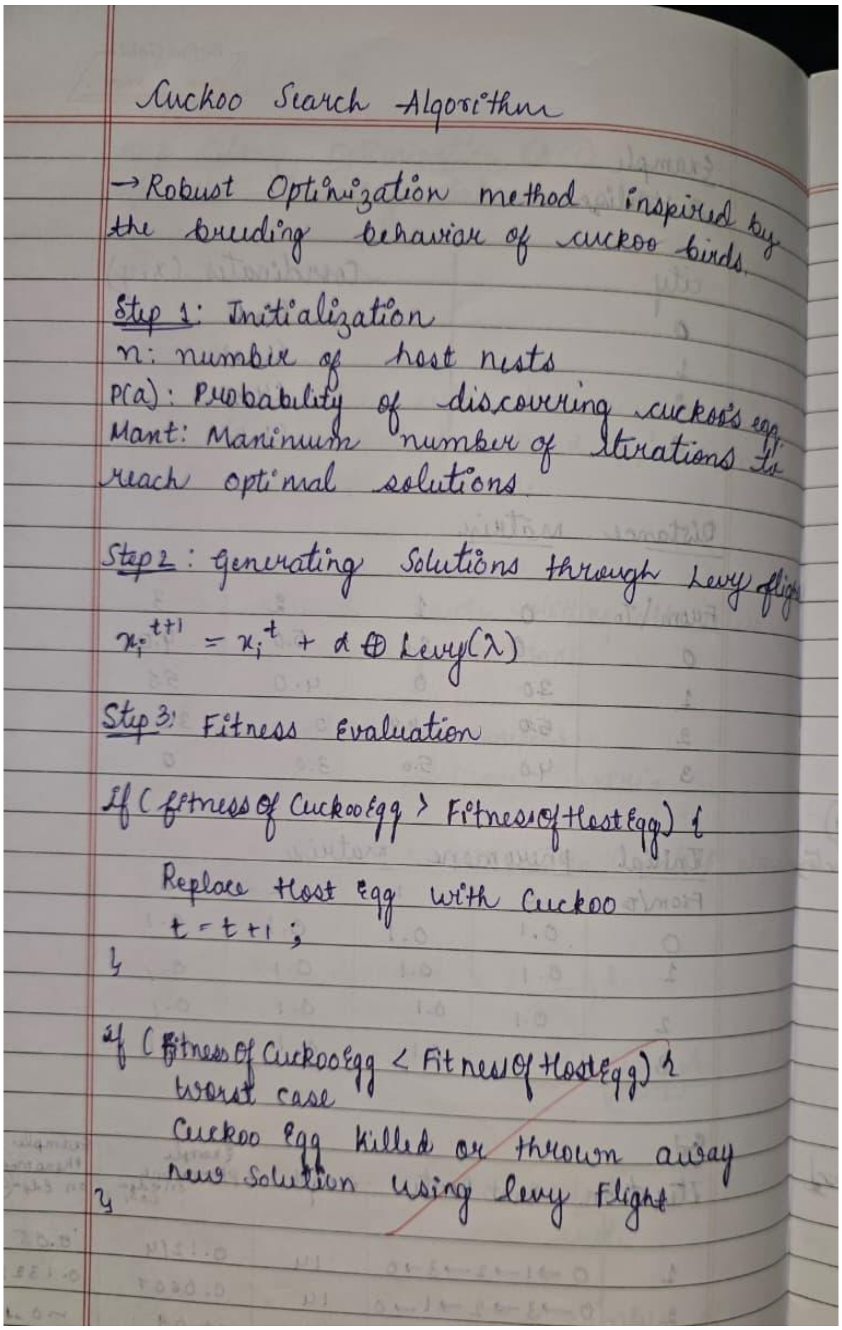
OUTPUT:

```
Iteration 1: Best Route = [0, 1, 2, 3], Length = 14.0
Iteration 2: Best Route = [0, 1, 2, 3], Length = 14.0
Iteration 3: Best Route = [0, 1, 2, 3], Length = 14.0
```

Program 4

Cuckoo Search (CS) Cuckoo Search Algorithms: We need to maximize the total value of selected items without exceeding the knapsack's weight capacity. Using the Cuckoo Search Algorithm, each solution is a binary vector, new solutions are generated via Lévy flights, and the best feasible solution is iteratively improved while abandoning poor solutions with a probability.

Observation:



Cuckoo Search Algorithm

→ Robust Optimization method inspired by the breeding behavior of cuckoo birds.

Step 1: Initialization

n: number of host nests

P(a): Probability of discovering cuckoo's egg

Maxt: Maximum number of iterations to reach optimal solutions

Step 2: Generating Solutions through Levy flight

$x_i^{t+1} = x_i^{t} + \alpha \oplus Levy(\lambda)$

Step 3: Fitness Evaluation

```
If (fitness of CuckooEgg > FitnessOfHostEgg) {
    Replace Host egg with Cuckoo
    t = t+1;
}

if (fitness of CuckooEgg < FitnessOfHostEgg) {
    worst case
    Cuckoo Egg killed or thrown away
    new Solution using levy Flight
}
```

Application

Q) Minimize $f(x) = x^2$
global minimum at $x=0$ $f(0) = 0$

Step 1

$x1 = 4$ $f(x1) = 16$
$x2 = -3$ $f(x2) = 9$
$x3 = 6$ $f(x3) = 36$

Best Sol^n
$x2 = -3$, fitness $= 9$

Step 2

$x_{new} = x_{old} + \alpha \cdot Levy(\lambda)$
$\alpha = 1$
$x_1^{new} = 4 + (-2) = 2$ $f(2) = 4$
$x_2^{new} = -3 + (1) = -2$ $f(-2) = 4$
$x_3^{new} = 6 + (-4) = 2$ $f(2) = 4$

Best sol^n $x = 2$ $f(2) = 4$

Step 3

Evaluate and Select Best

Nest 1 = 2 (fitness 4)
Nest 2 = -2 (fitness 4)
Nest 3 = 2 (fitness 4)

Step 4

$Pa = 0.25$
$x3 = -1$ $f(-1) = 1$
$x = -1$ $f(x) = 1$

Step 5:

$x = 0.5 \rightarrow f(x) = 0.25$

$x = 0 \quad f(x) = 0$

Global minimum found!

Eg

(1) $f(x) = x_1^2 + x_2^2$

Number of nests = 25

discovery rate = 0.25

100 iterations

25 nests in range [-5, 5]

Problem: Welded - Beam Design

minimize fabrication cost (material + welding) subject to constraints (stress, buckling, deflection, geometry).

$x_1 = h$ (weld thickness) (in)

$x_2 = l$ (weld length) (inches)

$x_3 = t$ = beam height

$x_4 = b$ = beam width

Minimize $f(x) = 1.10471\, x_1^2 x_2 + 0.04811\, x_3 x_4 (14 + x_2)$

(1) Initialize.

Set parameters n, P_a, T, λ, M.

penalized fitness

$$F(x) = f(x) + M \cdot \sum_i \max(0, g_i(x))^2$$

record x_{best}.

(2) loop $t = 1$ to T

Generate new solutions by Levy flights.

(3) New candidate: $y = x_i + \alpha \cdot s \odot (x_i - x_{best})$

step scale

elementwise multiply

(4) Evaluate candidates

$F(y) < F(x_i)$

select worst $k = \lceil p_\alpha \cdot n \rceil$ nests

(5) update best.

(6) ~~Adapt~~ stopping check

stop when $t \geq T$

MG 17/10/25

Code:

```
import numpy as np



def f(x):
    return x**2



def levy_flight():
    return np.random.randn()
```

```python
def cuckoo_search(n=3, pa=0.25, iterations=5):
    nests = np.array([4, -3, 6], dtype=float)
    best = nests[np.argmin(f(nests))]

    for t in range(iterations):
        for i in range(n):

            step = levy_flight()
            new = nests[i] + 1 * step


            if f(new) < f(nests[i]):
                nests[i] = new


        for i in range(n):
            if np.random.rand() < pa:
                nests[i] = np.random.uniform(-5, 5)

        best = nests[np.argmin(f(nests))]
        print(f"Iteration {t+1} | Best = {best}, f(x) = {f(best)}")

    return best


best_solution = cuckoo_search()
print("Final Best:", best_solution)
```

Output:

```
Iteration 1 | Best = -3.6431435629308795, f(x) = 13.272495020124703
Iteration 2 | Best = 2.189550165039872, f(x) = 4.794129925226131
Iteration 3 | Best = -0.2714863685552089, f(x) = 0.07370484831129473
Iteration 4 | Best = -0.2714863685552089, f(x) = 0.07370484831129473
Iteration 5 | Best = 0.2459712211369125, f(x) = 0.06050184162758391
Final Best: 0.2459712211369125
```

Program 5

Grey Wolf Optimizer (GWO) Using the Grey Wolf Optimizer (GWO), we aim to find the shortest, obstacle-free path by modeling the search agents (wolves) to iteratively converge toward the best position (path node) in the environment. The algorithm simulates the grey wolves' hunting hierarchy and encircling behavior to efficiently navigate the space from the start point.

Algorithm:

Gray wolf optimizer

Population based metaheuristic that mimics the pack hierarchy and hunting tactics of grey wolves.

Algorithm:-

1. define objective $f(x)$ fitness function
2. Set Parameters
   no. of wolves N
   iterations T
3. Sample N random positions within bounds
   α (best), β (2nd), δ (3rd)
4. $t = 1 \dots T$
   $a = 2 - 2 \cdot t / T$

   Generate random vectors
   $r_1, r_2 \sim U(0, 1)$
   $A = 2 a r_1 - a, \quad C = 2 r_2$
   $\alpha: D_\alpha - X_1$
   $X_\alpha - A \cdot D_\alpha$
   Repeat for β & δ to get X_2, X_3.
   $x = (X_1 + X_2 + X_3) / 3$
5. Stop at T iteration or earlier if best fitness stalls

→ Application
Continuous, nonlinear, multimodal problem

Bafna Gold
Date: Page:

Economic Load Dispatch

→ how much power each generator should produce. ~~to~~ so that
→ Total power demand is satisfied
→ cost of fuel is mimized
→ generators operate within min -max

Objective function
→ Minimize Fuel Cost

$C_i(P_i) = a_i P_i^2 + b_i P_i + c_i$

Total Cost:

$F = \sum_{i=1}^{n} C_i(P_i)$

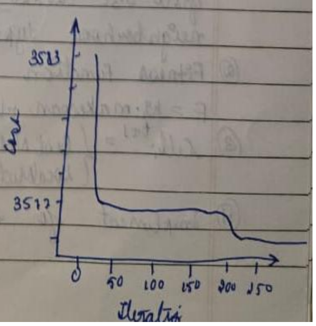
→

1] Initialization
N = 30-40
T = 500

2] Fitness evaluation
→ generator limit
→ power balance
→ total fuel cost + penalties
α → best β → second best δ → third best.

3] $X_{new} = \frac{X_1 + X_2 + X_3}{3}$

4] Apply constraints.
5] Terminate



Code:

```python
import numpy as np



a = np.array([0.003, 0.005, 0.001])
b = np.array([7, 8, 6])
c = np.array([100, 120, 150])

Pmin = np.array([50, 50, 50])
Pmax = np.array([200, 150, 180])

Pd = 350


```

```python
def cost(P):
    return np.sum(a * P**2 + b * P + c)



def penalty(P):
    total = np.sum(P)
    return 1000 * (abs(total - Pd))


def fitness(P):
    return cost(P) + penalty(P)



def GWO(num_wolves=30, max_iter=200):
    dim = 3
    lb, ub = Pmin, Pmax

    wolves = np.random.uniform(lb, ub, (num_wolves, dim))

    alpha, beta, delta = None, None, None

    for t in range(max_iter):
        for i in range(num_wolves):
            f = fitness(wolves[i])


            if alpha is None or f < fitness(alpha):
                delta = beta
                beta = alpha
                alpha = wolves[i].copy()
            elif beta is None or f < fitness(beta):
                delta = beta
                beta = wolves[i].copy()
            elif delta is None or f < fitness(delta):
                delta = wolves[i].copy()


        a_t = 2 - 2 * (t / max_iter)


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

```
                A1 = 2 * a_t * r1 - a_t
                C1 = 2 * r2

                D_alpha = abs(C1 * alpha[j] - wolves[i][j])
                X1 = alpha[j] - A1 * D_alpha

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

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

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


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

        if t % 50 == 0:
            print(f"Iteration {t} | Best Cost: {cost(alpha)}")

    return alpha, cost(alpha)



best_P, best_cost = GWO()
print("\nBest Power Output:", best_P)
print("Minimum Cost:", best_cost)
```

Output:

```
Iteration 0 | Best Cost: 2975.9786281503466
Iteration 50 | Best Cost: 2864.070964959683
Iteration 100 | Best Cost: 2864.070964959683
Iteration 150 | Best Cost: 2864.070964959683

Best Power Output: [ 53.18357901 116.81895164 180.
Minimum Cost: 2865.955482677795
```

Program 6

Parallel Cellular Algorithms and Programs The task is to perform edge detection or noise reduction in an image using Parallel Cellular Automata (PCA), where each pixel (cell) interacts with its neighbors to enhance edges or reduce noise iteratively.

Algorithm:

Parallel Cellular Algorithm

Algorithms inspired by the biological functioning of cells

Steps for Implementation

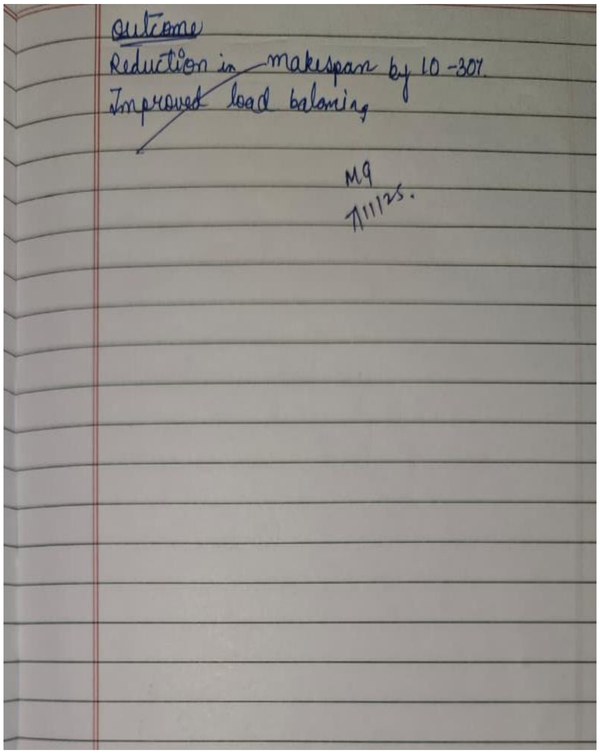
1. Define a problem →
   create function to optimize
2. Initialize Parameters →
   Set grid size, number of cells and iterations
3. Initialize population:
4. Evaluate Fitness:
   Access how well each cell performs
5. Update States
   → update based on neighbor interactions
6. Iterate until convergence
7. Track and report best solution

Application Problem

Parallel cellular Algorithm for Optimal task Scheduling in cloud Computing

1. Define how each cell represents a task-VM assignment solution.
   grid size (20×20)
   neighborhood type
2. Fitness Function.

   $F = w_1 \cdot \text{makespan} + w_2 \cdot \text{Load Variance} + w_3 \cdot \text{Energy Consumption}$
3. $\text{Cell}_i^{t+1} = \begin{cases} \text{Best Neighbour Solution} & \text{if better fitness} \\ \text{Local Mutation} & \text{otherwise} \end{cases}$
4. Impliment for 200 iterations



Outcome
Reduction in makespan by 10-30%
Improved load balancing

M9
7/11/25.

Code:

```
import numpy as np

GRID_X, GRID_Y = 20, 20
num_tasks = 10
num_vms = 4
w1, w2, w3 = 0.5, 0.3, 0.2

def fitness(x):
    loads = np.zeros(num_vms)
    for i in range(num_tasks):
        loads[x[i]] += np.random.randint(1, 10)
    return w1*np.max(loads) + w2*np.var(loads) + w3*(np.sum(loads)*0.1)

def neighbors(x, y):
    r=[]
    for dx,dy in [(1,0),(-1,0),(0,1),(0,-1)]:
        nx,ny=x+dx,y+dy
        if 0<=nx<GRID_X and 0<=ny<GRID_Y:
            r.append((nx,ny))
    return r

def PCA(iters=200):
```

```python
    G={}
    for i in range(GRID_X):
        for j in range(GRID_Y):
            G[(i,j)] = np.random.randint(0,num_vms,num_tasks)

    for t in range(iters):
        NG={}
        for i in range(GRID_X):
            for j in range(GRID_Y):
                c = G[(i,j)]
                bf = fitness(c)
                b = c
                for nx,ny in neighbors(i,j):
                    f = fitness(G[(nx,ny)])
                    if f < bf:
                        b = G[(nx,ny)]
                        bf = f
                NG[(i,j)] = b
        G = NG
    return b, bf

sol, fit = PCA()
print("Best Solution:", sol)
print("Best Fitness:", fit)
```

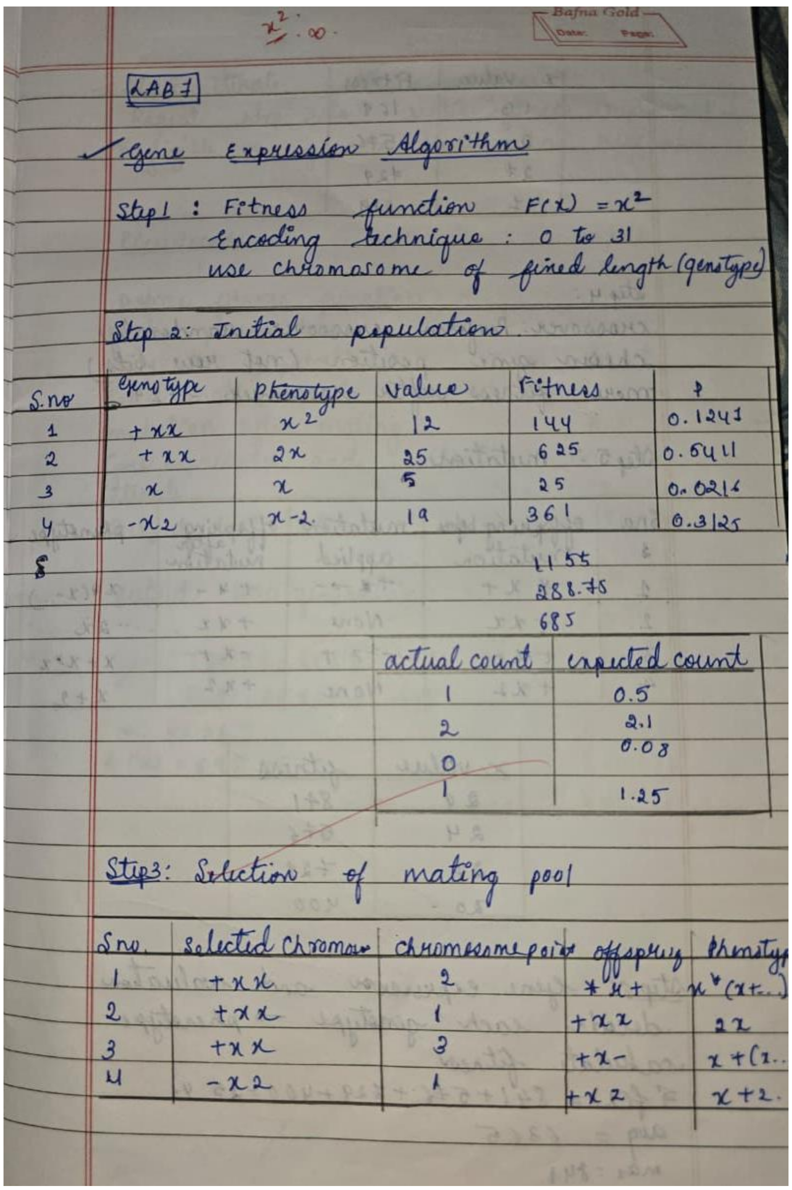
Output:

```
Best Solution: [1 1 0 2 3 2 1 3 0 2]
Best Fitness: 7.4399999999999995
```

Program 7

Optimization via Gene Expression Algorithms The Travelling Salesman Problem (TSP) asks for the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The provided text describes using a Genetic Algorithm to solve this by evolving city sequences (chromosomes) through selection, crossover, and mutation to minimize the total tour distance.

Algorithm:



LAB 7

Gene Expression Algorithm

Step 1 : Fitness function $F(x) = x^2$
Encoding technique : 0 to 31
use chromosome of fixed length (genotype)

Step 2: Initial population

S.no	Genotype	Phenotype	value	Fitness	P
1	$+xx$	x^2	12	144	0.1247
2	$+xx$	$2x$	25	625	0.5411
3	x	x	5	25	0.0216
4	$-x2$	$x-2$	19	361	0.3125
S				1155	
				288.75	
				685	

actual count	expected count
1	0.5
2	2.1
0	0.08
1	1.25

Step3: Selection of mating pool

Sno.	Selected Chromosome	chromosome pair	offspring	Phenotype
1	$+xx$	2	$*x+$	$x*(x+...)$
2	$+xx$	1	$+x2$	$2x$
3	$+xx$	3	$+x-$	$x+(x...$
4	$-x2$	1	$+x2$	$x+2...$

x value	Fitness
13	169
24	576
27	729
17	289

Step 4:
crossover: Perform crossover randomly chosen gene position (not raw bits)
more fitness after crossover = 729

Step 5: mutation

Sno.	offspring before mutation	mutation applied	offspring after mutation	phenotype
1	* x +	+ * → -	+ * -	x*(x-...)
2	+ x x	None	+ x x	2x
3	+ x -	- - → +	- x +	x + x*x
4	+ x 2	None	+ x 2	x + 2

x value	fitness
29	841
24	576
27	729
20	400

Step 6: Gene expression and evaluation
decode each genotype → phenotype
calculate fitness

$\Sigma f(x) = 841 + 576 + 729 + 400 = 2546$

avg = 636.5

max = 841

Bafna Gold
Date: Page:

Step 7: Iterate until convergence
Repeat step 3 to 6 until fitness improvement is negligible or generator limit has reached

Pseudocode:

Define fitness function
Define parameters
Generate population
Select mating pool
Mutation after mating
Gene expression and evaluation
Iterate
Output Best value

Output: 1000 generations

Genes: [29.53, 29.82, 29.84, 28.57, 15.09, 21.83, 23.03, 30.81, 28.51, 26.22]
x: 26.31
E(x) = 695.45

Code:

```
import random

def fitness(x):
    return x*x

def init_population():
    genes = ['+x', 'x', '2x', '-x']
    return random.sample(genes, 4)
```

```python
def express(gene, x):
    if gene == '+x': return x
    if gene == 'x': return x
    if gene == '2x': return 2*x
    if gene == '-x': return -x
    return x

def evaluate(pop):
    vals = [random.randint(1, 30) for _ in range(4)]
    phen = [express(pop[i], vals[i]) for i in range(4)]
    fit = [fitness(phen[i]) for i in range(4)]
    return vals, phen, fit

def select_mating(pop, fit):
    idx = sorted(range(4), key=lambda i: fit[i], reverse=True)
    return [pop[idx[0]], pop[idx[1]], pop[idx[2]], pop[idx[3]]]

def crossover(g1, g2):
    p = 1
    return g1[:p] + g2[p:], g2[:p] + g1[p:]

def mutate(gene):
    ops = ['+x', 'x', '2x', '-x']
    if random.random() < 0.3:
        return random.choice(ops)
    return gene

def gene_expression_algorithm(generations=10):
    pop = init_population()
    for gen in range(1, generations+1):
        vals, phen, fit = evaluate(pop)
        mating = select_mating(pop, fit)
        c1, c2 = crossover(mating[0], mating[1])
        c3, c4 = crossover(mating[2], mating[3])
        c1, c2, c3, c4 = mutate(c1), mutate(c2), mutate(c3), mutate(c4)
        pop = [c1, c2, c3, c4]
        best = max(fit)
        print(f"Generation {gen}: Fitness = {best}")
    return pop

result = gene_expression_algorithm()
print("\nFinal Genes:", result)
```

Output:

```
Generation 1: Fitness = 3600
Generation 2: Fitness = 400
Generation 3: Fitness = 484
Generation 4: Fitness = 324
Generation 5: Fitness = 841
Generation 6: Fitness = 900
Generation 7: Fitness = 784
Generation 8: Fitness = 441
Generation 9: Fitness = 625
Generation 10: Fitness = 576

Final Genes: ['x', 'x', '+x', '+x']
```