VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Vinayak Sunil Rodd (1WA23CS045)

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 **Vinayak Sunil Rodd (1WA23CS045),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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 Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

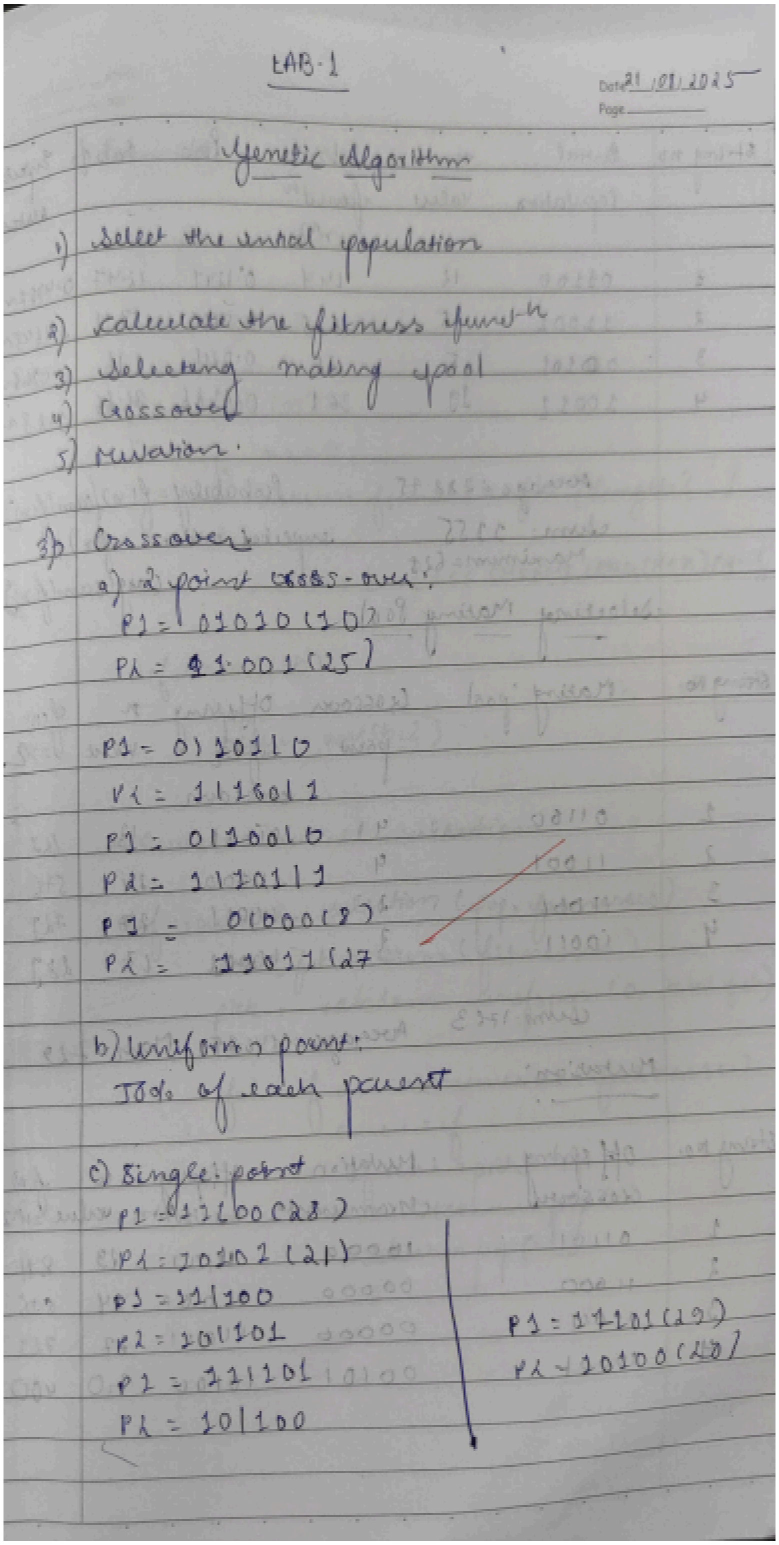
Github Link:

https://github.com/Vinayak1205/BISLAB/tree/main

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:



LAB-1

Date 21/08/2025

Page

Genetic Algorithm

1) Select the initial population

2) calculate the fitness func^n

3) Selecting mating pool

4) Crossover

5) Mutation.

3) Crossover:

a) 2 point cross-over:

P1 = 01010 (10)

P2 = 11001 (25)

P1 = 0110110

P2 = 1110011

P1 = 0110010

P2 = 1110111

P1 = 01000 (8)

P2 = 11011 (27

b) Uniform point:

50% of each parent

c) Single point

P1 = 11100 (28)

P2 = 10101 (21)

P1 = 11|100

P2 = 10|101

P1 = 11|101

P2 = 10|100

P1 = 11101 (29)

P2 = 10100 (20)

String no.	Initial population	x value	Fitness function $=x^2$	Prob.	Prob%	Expected output
1	01100	12	144	0.1247	12.47	0.4987≈1
2	11001	25	625	0.5411	54.11	2.1645≈2
3	00101	5	25	0.0216	2.16	0.0866≈0
4	10011	19	361	0.3126	31.26	1.254

Average = 288.75 Probability = f(x)/(sum(f(x))

Sum: 1155 expected output = f(x)/

Maximum = 625 (avg(sum(f(x))

Selecting Mating Pool

String No.	Mating pool	Crossover point	Offspring after crossover	x value	f(x) $=x^2$
1	01100	4	01101	13	169
2	11001	4	11000	24	576
3	11001	2	11011	27	729
4	10011	2	10001	17	289

Sum: 1763 Average = 440.75 , Max = 729

Mutation:

String no.	Offspring crossover	Mutation chromosomes	Offspring mutation	x value	f(x) $=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4.	10001	00101	10100	20	400

Pseudo Code

```
def fitness(x):

        return x**2

def encode(x):
        if x>=0:
            return format(x, f'0{Length}')
        else:
            return bin(1<<CHROME_LENGTH+x)[2:]

def decode(b):

        return int(b,2)

# Roulette wheel selection

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

Date __/__/__

Page ____

```
# single-point crossover
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

# Mutation (bit flip)
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)

#Main
```

]

Date __/__/__

Page ______

Output

Initial population: ['01100', '10111', '00101', '10011']

[12, 23, 5, 19]

Generation 1

x=12, bin= 01100, fit=144, prob=0.136,

exp-count=0.54

x=23, bin= 10111, fit=529,

prob= 0.500, exp-count = 1.00

x=5, bin=00101, fit=25,

prob=0.024, exp-count = 0.09

x=19, bin= 10011, fit=361, prob=0.341,

exp-count = 1.36

Generation 2

x=23, bin=10111, fit=529, prob=0.309,

exp-count = 1.24

x=23, bin = 10111, fit =529, prob =0.30

exp-count = 1.14

x=13, bin = 01101, fit=169, prob=0.09

exp-count = 0.40

x=22, bin=10110, fit =484,

prob=0.283, exp-count = 1

Generation 3

x=6, bin=00110, fit=36, prob=0.023,
exp-count = 0.09
x=2 3, bin=10111, fit=529, prob=0.335,
~~x=22~~ exp-count = 1.34,
x=22, bin = 10110, fit=484,
prob = 0.307, exp-count = 1.23
x=23, bin = 10111, fit=529,
prob = 0.335, exp-count = 1.34

Generation 4

x=22 : bin = 10110, fit = 484, prob=0.171,
exp-count = 0.68
x=31, bin= 11111, fit=961, prob=0.340,
exp-count = 1.36
x=30, bin= 11110, fit = 900, prob = 0.318,
exp. count = 1.27
x=22, bin= 10110, fit = 484, prob = 0.171,
exp-count = 0.68.

Final best solution: 30 11110
fitness = 900

Code:

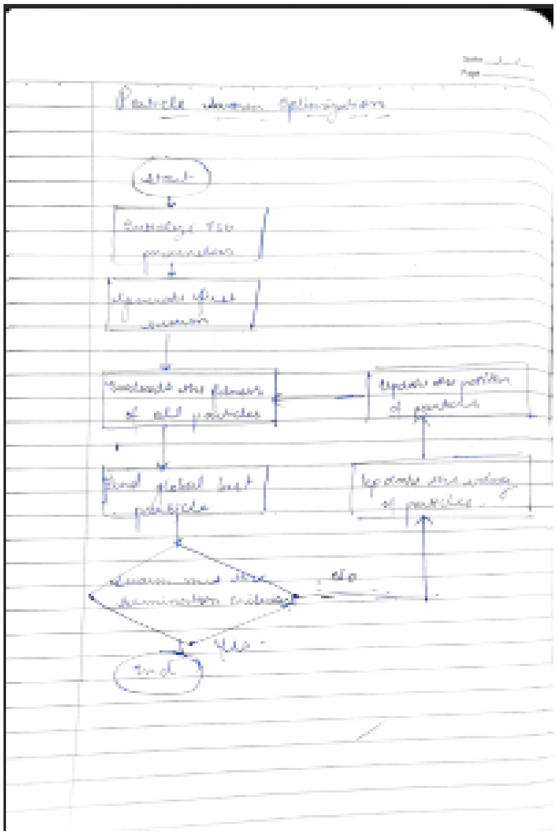
```
import random
import math
from dataclasses import dataclass
from typing import List, Tuple

# ------- Helpers -------
BIT_LEN = 5 # 5-bit chromosome
LOW, HIGH = 0, 31

def encode(x: int) -> str:
    return format(x, f"0{BIT_LEN}b")

def decode(bits: str) -> int:
    return int(bits, 2)

def fitness(x: int) ->
    int: return x * x

def single_point_crossover(p1: str, p2: str, point: int) -> Tuple[str, str]:
    return p1[:point] + p2[point:], p2[:point] + p1[point:]

def mutate(bits: str, rate: float) -> str:
    out = []
    for ch in bits:
        if random.random() < rate:
            out.append('1' if ch == '0' else '0')
        else:
            out.append(ch)
    return "".join(out)

# ---------- Roulette Selection ----------
def roulette_select(pop: List[str]) -> List[str]:
    xs = [decode(b) for b in pop]
    fs = [fitness(x) for x in xs]
    total = sum(fs)
    if total == 0:
        return random.choices(pop, k=len(pop))
    probs = [f / total for f in fs]
    # cumulative distribution
    cum = []
    acc = 0.0
    for p in
        probs: acc
        += p
        cum.append(acc)
    sel = []
    for _ in
        range(len(pop)): r =
        random.random()
        for i, c in enumerate(cum):
            if r <= c:
                sel.append(pop[i])
                break
    return sel

# ------- GA          -------
Config @dataclass
class GAConfig:
    population_size: int = 12
    bit_len: int = BIT_LEN
    crossover_rate: float =
    0.9
    mutation_rate: float = 1.0 / BIT_LEN
    generations: int = 20

# ---------- Main Evolution ----------
def evolve(config: GAConfig, seed_pop: List[str] = None) -> Tuple[List[str], List[Tuple[int,int]]]:
    if seed_pop is None:
        pop = [encode(random.randint(LOW, HIGH)) for _ in range(config.population_size)]
    else:
        pop = seed_pop[:]
        while len(pop) < config.population_size:
            pop.append(encode(random.randint(LOW, HIGH)))
        pop = pop[:config.population_size]

    history = []
```

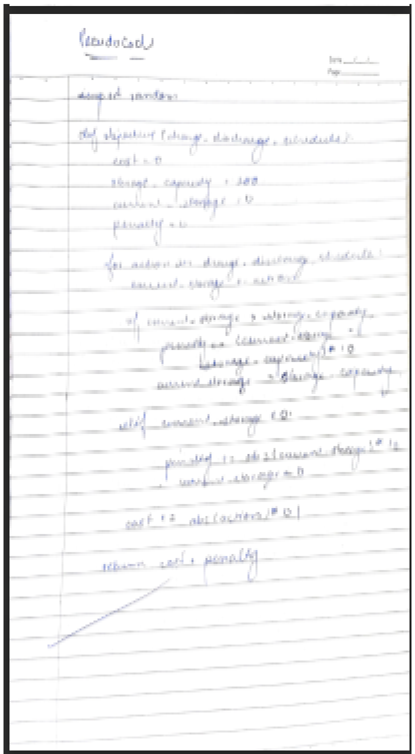
```
for g in range(config.generations):
        xs = [decode(b) for b in pop]
        fs = [fitness(x) for x in xs]
        best = max(zip(xs, fs), key=lambda t: t[1])
        history.append(best)

        # Selection
        mating = roulette_select(pop)

        # Crossover
        next_pop =
        []
        for i in range(0, config.population_size, 2):
            p1, p2 = mating[i], mating[i+1]
            if random.random() < config.crossover_rate:
                point = random.randint(1, config.bit_len - 1)
                c1, c2 = single_point_crossover(p1, p2, point)
                next_pop.extend([c1, c2])
            else:
                next_pop.extend([p1, p2])

        # Mutation
        next_pop = [mutate(b, config.mutation_rate) for b in next_pop]
        pop = next_pop

    # final best
    xs = [decode(b) for b in pop]
    fs = [fitness(x) for x in xs]
    best = max(zip(xs, fs), key=lambda t: t[1])
    history.append(best)
    return pop, history

# ------- Run -------
if _name____== "_main_":
    random.seed(42)
    cfg = GAConfig(population_size=12, generations=30)

    # initial population (optional, from board)
    init_bits = ["01100", "11001", "00101", "10011"]
    final_pop, hist = evolve(cfg, seed_pop=init_bits)
    print("Final Population:")
    for b in final_pop:
        x = decode(b)
        print(f"{b} -> x={x}, fitness={fitness(x)}")

    print("\nBest per generation:")
    for gen, (x, f) in enumerate(hist):
        print(f"Gen {gen}: x={x}, fitness={f}")
```

Program 2

Particle Swarm Optimization for Function Optimization

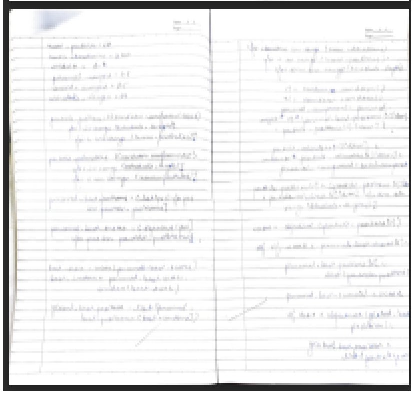
Algorithm:


Particle swarm Optimization
Start
Initialize PSO parameters
Generate first swarm
Evaluate the fitness of all particles
Update the position of particle
Find global best particle
Update the velocity of particle
Swarm meet the termination criteria
No
Yes
End



Pseudocode

```
import random

def objective(charge, discharge, schedule):
    cost = 0
    storage_capacity = 100
    current_storage = 0
    penalty = 0

    for action in charge, discharge, schedule:
        current_storage += action

        if current_storage > storage_capacity:
            penalty += (current_storage -
                   storage_capacity)** 2
            current_storage = storage_capacity

        elif current_storage < 0:

            penalty += abs(current_storage)** 2
            current_storage = 0

        cost += abs(action)** 2)

    return cost, penalty
```



[illegible]

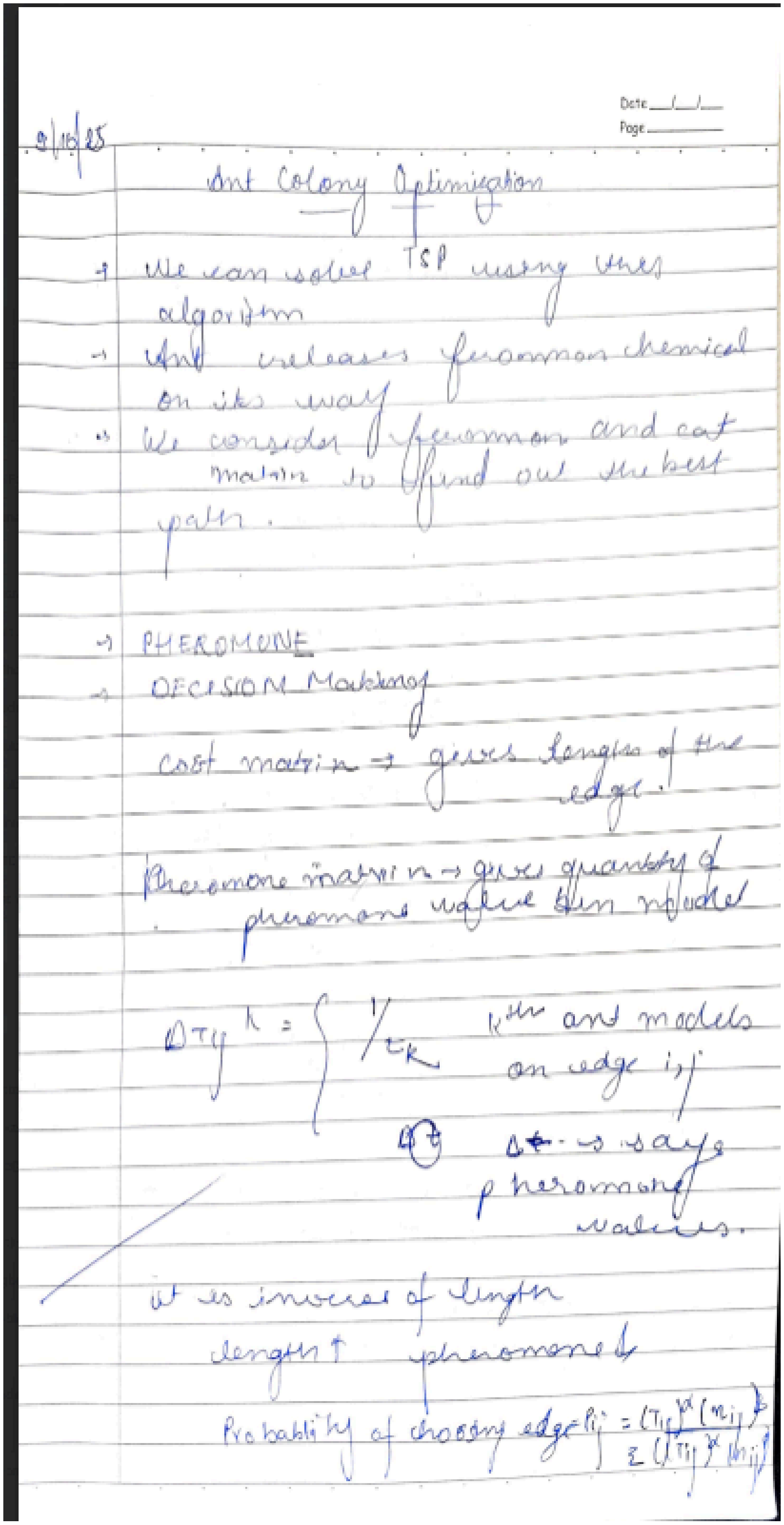
Code:

```
import random

# Objective function: power consumption = x^2 + y^2
def power_consumption(position):
    x, y = position
    return x**2 + y**2

# PSO parameters
num_particles =
30
num_iterations = 100
w = 0.5           # inertia weight
c1 = 1.5          # cognitive
coefficient c2 = 1.5            # social
coefficient

# Search space bounds
x_min, x_max = 0.1, 2.0    # thickness in
mm y_min, y_max = 1.0, 10.0 # length in
cm

# Initialize particles
particles = []
for _ in range(num_particles):
    position = [random.uniform(x_min, x_max), random.uniform(y_min, y_max)]
    velocity = [random.uniform(-1, 1), random.uniform(-1, 1)]
    particles.append({
        'position': position,
        'velocity': velocity,
        'best_position': position[:],
        'best_score': power_consumption(position)
    })

# Find global best
global_best = min(particles, key=lambda p: p['best_score'])
global_best_position = global_best['best_position'][:]
global_best_score = global_best['best_score']

# PSO loop
for iteration in range(num_iterations):
    for particle in particles:
        # Update velocity
        for i in range(2): # x and y
            r1, r2 = random.random(), random.random()
            cognitive = c1 * r1 * (particle['best_position'][i] - particle['position'][i])
            social = c2 * r2 * (global_best_position[i] - particle['position'][i])
            particle['velocity'][i] = w * particle['velocity'][i] + cognitive + social

        # Update position
        for i in range(2):
            particle['position'][i] +=
            particle['velocity'][i] # Clamp position to
            bounds
            if i == 0:
                particle['position'][i] = max(x_min, min(x_max, particle['position'][i]))
            else:
                particle['position'][i] = max(y_min, min(y_max, particle['position'][i]))

        # Evaluate fitness
        score = power_consumption(particle['position'])

        # Update personal best
        if score < particle['best_score']:
            particle['best_position'] = particle['position'][:]
            particle['best_score'] = score

            # Update global best
            if score < global_best_score:
                global_best_position = particle['position'][:]
                global_best_score = score

# Output result
print("Best design found:")
print(f" Filament thickness (x): {global_best_position[0]:.4f} mm")
print(f" Filament length     (y): {global_best_position[1]:.4f} cm")
```

PROGRAM 3

Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:



Date__/__/__
Page____

9/10/25

Ant Colony Optimization

→ We can solve TSP using this algorithm

→ Ant releases feromon chemical on its way

→ We consider feromon and cost matrix to find out the best path.

→ PHEROMONE

→ DECISION Making

cost matrix → gives lengths of the edge.

Pheromone matrix → gives quantity of pheromone value then update

$\Delta\tau_{ij}^{k} = \begin{cases} 1/L_k & k^{th} \text{ ant models on edge } i,j \\ 0 & \end{cases}$

Δt → says pheromone values.

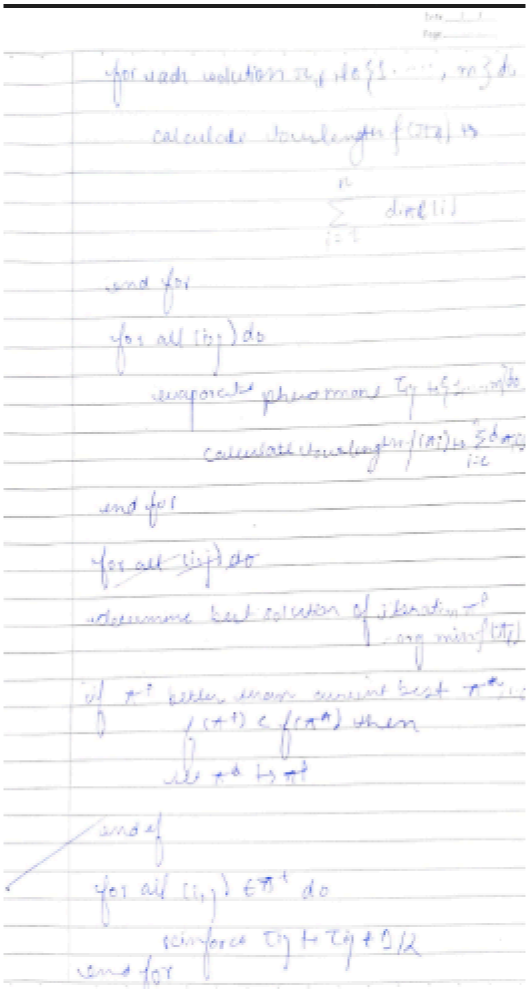
it is inverse of length

length ↑ pheromone ↓

Probability of choosing edge $P_{ij} = \dfrac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\Sigma((\tau_{ij})^{\alpha}(\eta_{ij})^{\beta})}$

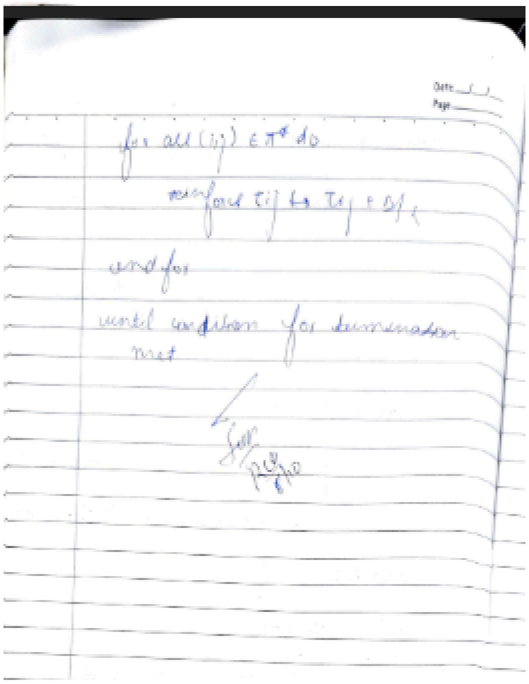
Algorithm

```
initialize pheromone values ∀ i,j ∈ Q
    : $\tau_{ij} \mapsto \tau_0$

repeat
    for each ant $l = \{1 \ldots m\}$ do
        initialize selection set $S \mapsto S \setminus \{i\}$
        randomly ~~choose~~ starting
        city $i_0 \in S$ for ant $l$
        move
        move to starting city $i \mapsto i_0$

    while $S \neq \phi$ do

        remove current city from
            selection set $S \mapsto S \setminus \{i\}$

        choose next city j in tour
        with probability $p_{ij}$
            $= \dfrac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in S} \tau_{ih}^{\alpha} \cdot \eta_{ih}^{\beta}}$

        update solution vector $\pi_l(i) \mapsto j$
        move to new city $i \mapsto j$

    end while
    finalize solution vector $\pi_l(i) \mapsto i_0$

end for
```



Date: / /
Page:

```
for each solution $\pi_k$, $k \in \{1 \cdots, m\}$ do
    calculate tourlength $f(\pi_k)$ is
        $\sum_{i=1}^{n} dist(i)$
end for
for all (i,j) do
    evaporate pheromone $\tau_{ij}$ $k \in \{1 \cdots m\}$ do
        calculate tourlength $f(\pi_k) = \sum_{i=1} d_{\pi(i)}$
end for
~~for all (i,j) do~~
determine best solution of iteration $\pi^t$
    = arg min $f(\pi_k)$
if $\pi^t$ better than current best $\pi^*$, i.e.
    $f(\pi^t) < f(\pi^*)$ then
    set $\pi^* \leftarrow \pi^t$
end if
for all $(i,j) \in \pi^t$ do
    reinforce $\tau_{ij} \leftarrow \tau_{ij} + Q/2$
end for
```



Date: / /
Page:

```
for all $(i,j) \in \pi^*$ do
    reinforce $\tau_{ij}$ to $\tau_{ij} + Q/2$
end for
until condition for termination
    met
```

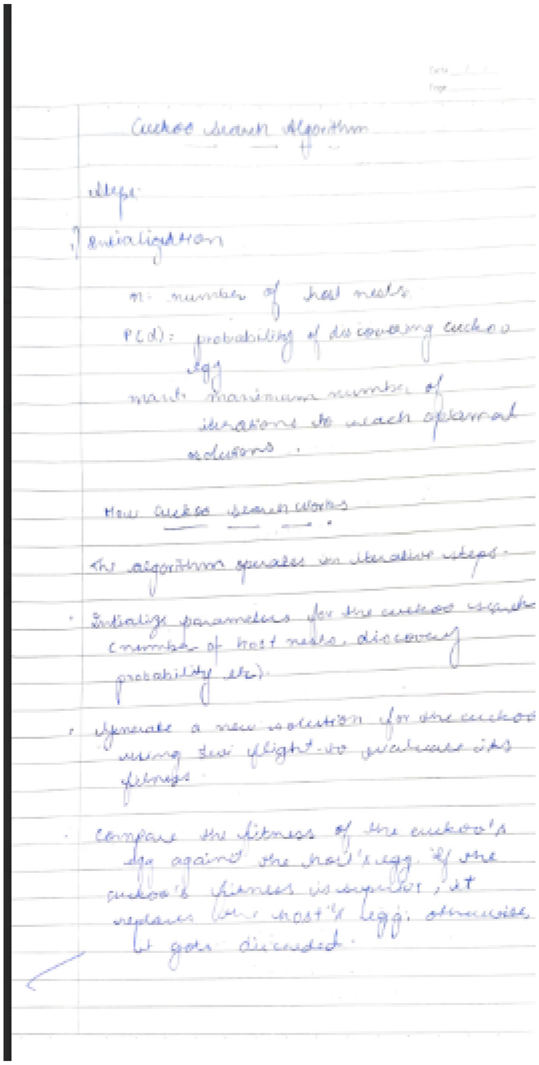
Code:

```
import numpy as np

# Distance between cities
def distance(city1, city2):
    return np.linalg.norm(np.array(city1) - np.array(city2))

# Initialize pheromone levels
def initialize_pheromones(num_cities, initial_pheromone):
    return np.full((num_cities, num_cities),
    initial_pheromone)

# Choose next city based on pheromone and heuristic info
def choose_next_city(current_city, unvisited, pheromone, distances, alpha, beta):
    pheromone_vals = pheromone[current_city, unvisited] ** alpha
    heuristic_vals = (1 / distances[current_city, unvisited]) ** beta
    probs = pheromone_vals * heuristic_vals
    probs /= probs.sum()
    return np.random.choice(unvisited, p=probs)

# Compute total length of a tour
def tour_length(tour, distances):
    length = 0
    for i in range(len(tour) - 1):
        length += distances[tour[i], tour[i+1]]
    length += distances[tour[-1], tour[0]] # return to start
    return length

# Main ACO function
def ant_colony_optimization(cities, num_ants=10, num_iterations=100, alpha=1, beta=5,
evaporation=0.5, Q=100):
    num_cities = len(cities)
    distances = np.zeros((num_cities,
    num_cities)) for i in range(num_cities):
        for j in range(num_cities):
            distances[i][j] = distance(cities[i], cities[j])
    pheromone = initialize_pheromones(num_cities, initial_pheromone=1.0)
    best_tour = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        all_tours = []
        all_lengths = []

        for ant in
            range(num_ants): tour =
            []
            unvisited = list(range(num_cities))
            current_city = np.random.choice(unvisited)
            tour.append(current_city)
            unvisited.remove(current_city)

            while unvisited:
                next_city = choose_next_city(current_city, unvisited, pheromone, distances, alpha, beta)
                tour.append(next_city)
                unvisited.remove(next_city)
                current_city = next_city

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

            if length < best_length:
                best_length = length
                best_tour = tour
        # Evaporate pheromone
        pheromone *= (1 -
        evaporation)

        # Deposit pheromone proportional to quality
        for tour, length in zip(all_tours, all_lengths):
            deposit_amount = Q / length
            for i in range(num_cities - 1):
                a, b = tour[i], tour[i+1]
                pheromone[a][b] +=
                deposit_amount pheromone[b][a]
                += deposit_amount
            # Add pheromone for return edge
            a, b = tour[-1], tour[0]
            pheromone[a][b] +=
            deposit_amount pheromone[b][a]
            += deposit_amount

        if iteration % 10 == 0 or iteration == num_iterations - 1:
            print(f"Iteration {iteration+1}, best length: {best_length:.2f}")

    return best_tour, best_length

# Example usage:

cities = [
    (0, 0), (1, 5), (5, 2), (6, 6), (8, 3)
]
best_tour, best_length =
ant_colony_optimization(cities) print("Best tour:",
best_tour)
print("Best length:", best_length)
```

PROGRAM 4:

Cuckoo Search (CS)

ALGORITHM:



Cuckoo Search Algorithm

Steps:

1) Initialization

n: number of host nests

P(d): probability of discovering cuckoo egg

maxt: maximum number of iterations to reach optimal solutions.

How Cuckoo Search Works.

The algorithm operates in iterative steps.

- Initialize parameters for the cuckoo search (number of host nests, discovery probability etc).
- Generate a new solution for the cuckoo using Levi flight to evaluate its fitness.
- Compare the fitness of the cuckoo's egg against the host's egg. If the cuckoo's fitness is superior, it replaces the host's egg; otherwise it gets discarded.

- Generating solutions through Levy Flight

$$\underbrace{x_i^{t+1}}_{\text{New solution}} = \underbrace{x_i^{t}}_{\text{Current location}} + \underbrace{\alpha}_{\text{Step size}} \underbrace{\oplus}_{\text{Entry wise multiplication}} \text{Levy}(\overset{\text{Levy exponent}}{\lambda})$$

- Fitness Evaluation

It determines how suitable a solution (egg) is within its respective nest.

```
If (Fitness Of Cuckoo Egg > Fitness Of Host Egg)
{
    Replace Host Egg with Cuckoo's
    t = t+1;
}

If (Fitness of Cuckoo Egg < Fitness of Host Egg)
{
    Worst case
    Cuckoo Egg killed or throw away
    Generate new solution again using
    Levy Flight
}
```

Cuckoo Search Algorithm

```
1) Set the initial value of the host nest
   size n, probability $P_a \in (0,1)$ and
   max no. of iteration.

2) Set $t = 0$

3) For $(i = 1 : i \le n)$ do

4) Generate initial population of n host $x_i^t$

5) Evaluate fitness function $f(x_i^t)$.

6) End for.

7) Generate a new solution (cuckoos) $x_i^{t+1}$
   randomly by Levy flight

8) Evaluate fitness function $x_i^{t+1}$ i.e.,
   $f(x_i^{t+1})$

9) Choose a nest $x_j$ among n solutions
   random

10) If ( $f(x_i^{t+1}) > (f x_j^t)$ ) then

      replace the solution $x_j$ with solution
      $x_i^{t+1}$

11) End if.
```

13) Abandon a fraction Pa of worst nest

14) Build new nest at new location using Levy flight a fraction Pa (of worse nest)

15) Keep the best solution (nest with quality solution)

16) Rank the solution and find current best solution

17) Set t = t+1

18) Until (t ≥ Maxd)

19) Produce the best solution

Output = Best Nest X_i.

Date __/__/__

Page ______

Output:-

Iteration 0: Best distance = 51.00
Iteration 50: Best distance : 45.00
Iteration 100; Best distance : 45.00

Iteration 400 : Best distance = 45.00

Final Best route : [1, 4, 0, 5, 6, 7, 3, 2]
Final Best Distance : 45

CODE:

```
import numpy as np
import random

# Sigmoid activation function and its derivative for
neural network
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define a simple feedforward neural network
def neural_network(weights, inputs):
    input_layer = inputs

    hidden_layer = sigmoid(np.dot(input_layer,
weights['W1']) + weights['b1'])
    output_layer = sigmoid(np.dot(hidden_layer,
weights['W2']) + weights['b2'])
    return output_layer, hidden_layer

# Fitness function (Mean Squared Error)
def fitness_function(weights, inputs, outputs, target):
    predictions, _ = neural_network(weights, inputs)
    error = np.mean((predictions - target) ** 2)
    return error

# Initialize nests (cuckoos) with random weights
def initialize_nests(population_size, input_size,
hidden_size, output_size):
    nests = []
    for _ in range(population_size):
        nest = {
            'W1': np.random.randn(input_size, hidden_size),
            'b1': np.random.randn(hidden_size),
            'W2': np.random.randn(hidden_size,
output_size),
            'b2': np.random.randn(output_size)
        }
        nests.append(nest)
    return nests

# Lévy flight for generating new solutions
def levy_flight(current_nest,
levy_step_size):
    new_nest = {
        'W1': current_nest['W1'] + levy_step_size *
np.random.randn(*current_nest['W1'].shape),
        'b1': current_nest['b1'] + levy_step_size *
np.random.randn(*current_nest['b1'].shape),
        'W2': current_nest['W2'] + levy_step_size *
np.random.randn(*current_nest['W2'].shape),
        'b2': current_nest['b2'] + levy_step_size *
np.random.randn(*current_nest['b2'].shape)
    }
    return new_nest


# Replace a fraction of nests with random solutions
def replace_nests_with_random_discovery(nests,
discovery_rate, input_size, hidden_size, output_size):
```

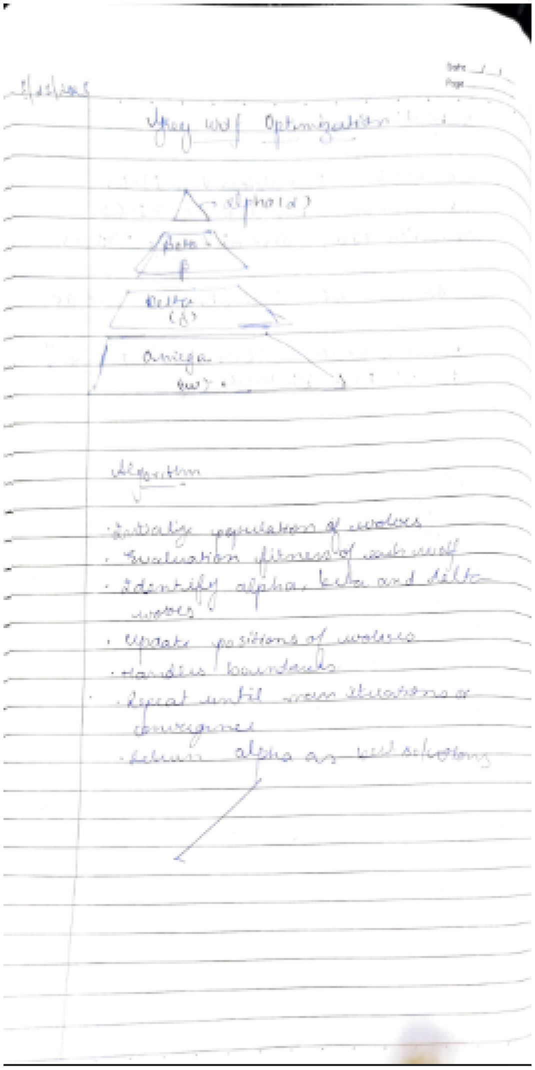
```
num_replace = int(discovery_rate * len(nests))
for i in range(num_replace):
    nests[i] = {
        'W1': np.random.randn(input_size, hidden_size),
        'b1': np.random.randn(hidden_size),
            'W2': np.random.randn(hidden_size,
output_size),
            'b2': np.random.randn(output_size)
        }
    return nests

# Cuckoo Search main function
def cuckoo_search(inputs, target, population_size,
max_iterations, discovery_rate, levy_step_size):
    input_size = inputs.shape[1] # Number of input
features
    hidden_size = 5 # Hidden layer size (can be adjusted)
    output_size = target.shape[1] # Number of output
neurons (1 for regression or number of classes for
classification)

    # Initialize nests (cuckoos)
    nests = initialize_nests(population_size, input_size,
hidden_size, output_size)

    # Track the best nest
    best_nest = nests[0]
    best_fitness = fitness_function(best_nest, inputs,
target, target)

    # Main loop of Cuckoo Search
    for iteration in range(max_iterations):
        for i in range(population_size):
            # Evaluate fitness of the current nest (cuckoo)
            fitness = fitness_function(nests[i], inputs, target,
target)

            # If the fitness is better, update the best solution
            if fitness < best_fitness:
                best_nest = nests[i]
                best_fitness = fitness

            # Generate new candidate solution via Lévy
flight
            new_nest = levy_flight(nests[i], levy_step_size)

            # Evaluate fitness of the new solution
            new_fitness = fitness_function(new_nest, inputs,
target, target)

            # If the new solution is better, replace the old one
            if new_fitness < fitness:
                nests[i] = new_nest

        # Replace some nests with random solutions
(discovery rate)
```

```
        nests =
replace_nests_with_random_discovery(nests,
discovery_rate, input_size, hidden_size, output_size)

        # Print the progress
        print(f"Iteration {iteration + 1}/{max_iterations},
Best Fitness: {best_fitness}")

    return best_nest

# Main function to run the neural network training with
Cuckoo Search
if _name____== "_main_": #
    User-defined parameters
    population_size = int(input("Enter population size: "))
    max_iterations = int(input("Enter max iterations: "))
    discovery_rate = float(input("Enter discovery rate
(between 0 and 1): "))
    levy_step_size = float(input("Enter Levy step size: "))

    # Example data (XOR problem, you can replace with
your data)
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    target = np.array([[0], [1], [1], [0]]) # XOR outputs

    # Train the neural network using Cuckoo Search
    best_weights = cuckoo_search(inputs, target,
population_size, max_iterations, discovery_rate,
levy_step_size)

    # Final trained model's weights
    print("\nFinal Trained Weights:")
    print("W1:", best_weights['W1'])
    print("b1:", best_weights['b1'])
    print("W2:", best_weights['W2'])
    print("b2:", best_weights['b2'])

    # Optionally test the model
    predictions, _ = neural_network(best_weights, inputs)
    print("\nPredictions on the training set:")
    print(predictions)
```

PROGRAM 5:

Grey Wolf Optimizer (GWO)

Algorithm:



Date / /
Page

4/11/24

Grey Wolf Optimization

alpha (α)
Beta
β
Delta
(δ)
omega
(ω)

Algorithm

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

Date __/__/__
Page ______

Mathematical Model

The core of GWO model involves updating the positions of the search agents and the search space.

D: Distance to prey $D = |C \cdot X_p - X|$

where X_p is the position of the prey (the best solution found so far, e.g. alpha wolf)

X is the current position of the wolf
C is a coefficient vector that introduces stochastic behaviour and can be calculated as $C = 2 \cdot r_2$.

where r_2 is a random number between 0 and 1

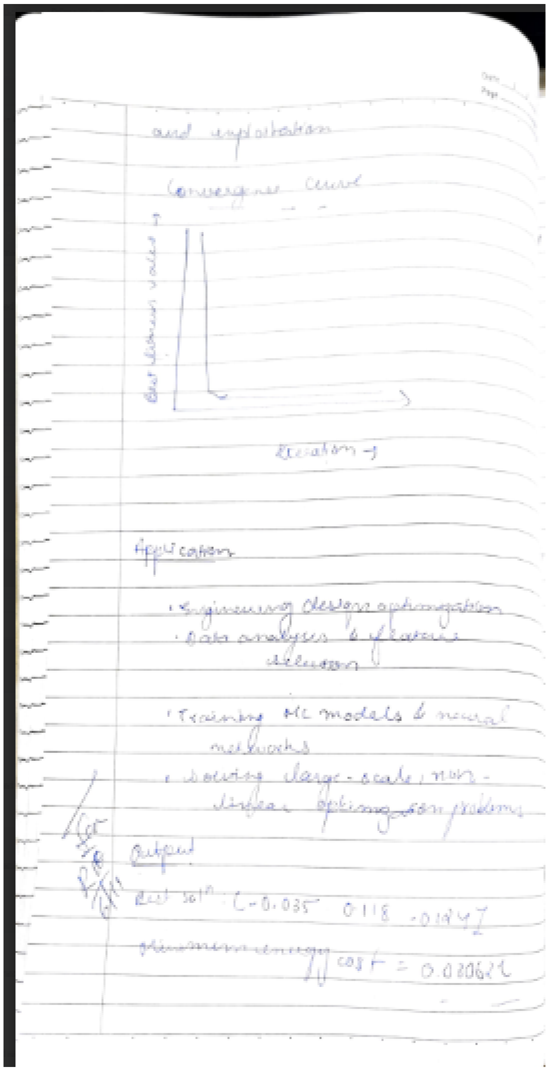
Wolf position update (X):

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

where A is another coefficient vector.

$$A = 2a\,(r_1 - a)$$

where a is a vector that decreases linearly from 2 to 0 over the iterations, controlling the balance between exploration



and exploitation

Convergence curve

Best fitness value →
Iteration →

Application

- Engineering design optimization
- Data analysis & feature selection
- Training ML models & neural networks
- Solving large-scale, non-linear optimization problems

Output

Best soln [-0.035 0.118 -0.194]

~~Minimum energy~~ cost = 0.020622

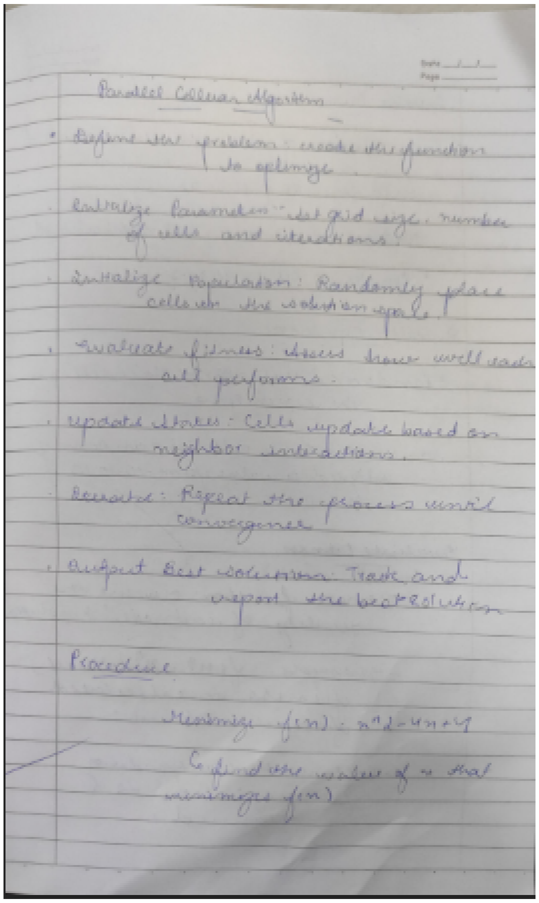
CODE:

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import
train_test_split from sklearn.tree import
DecisionTreeClassifier from sklearn.metrics import
accuracy_score

# ------------------
# Fitness
Function #-----------
def fitness_function(features, X_train, X_test, y_train, y_test):
    # If no features selected → very bad fitness
    if np.sum(features) == 0:
        return 1e9
    # Select features
    X_train_sel = X_train[:, features == 1]
    X_test_sel = X_test[:, features == 1]

    # Train & evaluate classifier
    clf = DecisionTreeClassifier()
    clf.fit(X_train_sel, y_train)
    y_pred =
    clf.predict(X_test_sel)
    acc = accuracy_score(y_test, y_pred)

    # Fitness: lower is better
    return (1 - acc) + (np.sum(features) / len(features))

# ------------------
# Grey Wolf Optimization
# ------------------
def GWO(num_wolves, max_iter, num_features, X_train, X_test, y_train, y_test):
    # Initialize wolves randomly (binary vectors)
    wolves = np.random.randint(0, 2, (num_wolves, num_features))

    # Evaluate fitness
    fitness = np.array([fitness_function(w, X_train, X_test, y_train, y_test) for w in wolves])

    # Identify alpha, beta, delta
    sorted_idx = np.argsort(fitness)
    alpha, beta, delta = wolves[sorted_idx[:3]]

    for t in range(max_iter):
        a = 2 - 2 * (t / max_iter) # Decreasing from 2 to 0

        for i in range(num_wolves):
            if np.array_equal(wolves[i], alpha) or np.array_equal(wolves[i], beta) or
np.array_equal(wolves[i], delta):
                continue

            # Position update based on alpha, beta, delta
            for j in range(num_features):
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2*a*r1 - a, 2*r2
                D_alpha = abs(C1 * alpha[j] - wolves[i][j])
                X1 = alpha[j] - A1 * D_alpha

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

                r1, r2 = np.random.rand(), np.random.rand()
                A3, C3 = 2*a*r1 - a, 2*r2
                D_delta = abs(C3 * delta[j] - wolves[i][j])
                X3 = delta[j] - A3 * D_delta
                wolves[i][j] = 1 if ((X1 + X2 + X3) / 3) > 0.5 else
        0 # Re-evaluate fitness
        fitness = np.array([fitness_function(w, X_train, X_test, y_train, y_test) for w in wolves])
        sorted_idx = np.argsort(fitness)
        alpha, beta, delta = wolves[sorted_idx[:3]]

    return alpha

# ------------------
# Run Example
# ------------------
if _name____== "_main_": #
    Load dataset
    data = load_breast_cancer()
    X_train, X_test, y_train, y_test = train_test_split(
        data.data, data.target, test_size=0.3, random_state=42
    )

    # Run GWO
    best_features = GWO(num_wolves=10, max_iter=20,
                        num_features=X_train.shape[1], X_train=X_train, X_test=X_test,
                        y_train=y_train, y_test=y_test)

    print("Selected features:", np.where(best_features == 1)[0])
```

PROGRAM 6:

Parallel Cellular Algorithms and Programs

ALGORITHM:



Date __/__/__
Page ____

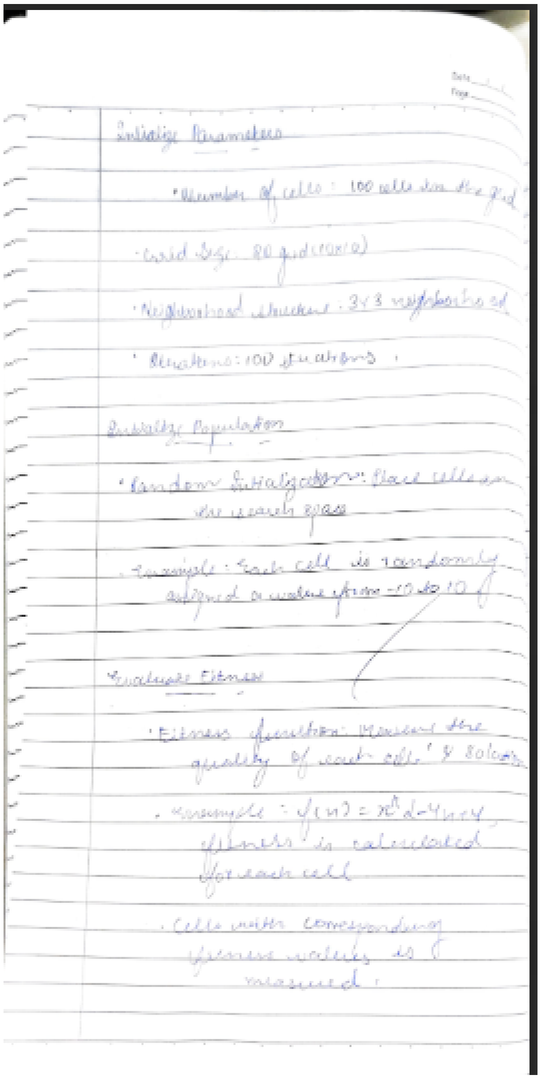
Parallel Cellular Algorithm

- Define the problem: create the function to optimize.
- Initialize Parameters: set grid size, number of cells and iterations.
- Initialize Population: Randomly place cells in the solution space.
- Evaluate fitness: Assess how well each cell performs.
- Update States: Cells update based on neighbor interactions.
- Iterate: Repeat the process until convergence
- Output Best solution: Track and report the best solution

Procedure

Minimize $f(x) = x^2 - 4x + 4$

↳ find the value of x that minimizes f(x)



Date ___ / /
Page ___

Initialize Parameters

· Number of cells: 100 cells in the grid

· Grid Size: 80 grid(10X10)

· Neighborhood structure: 3x3 neighborhood

· Iterations: 100 iterations.

Initialize Population

· Random Initialization: Place cells in the search space

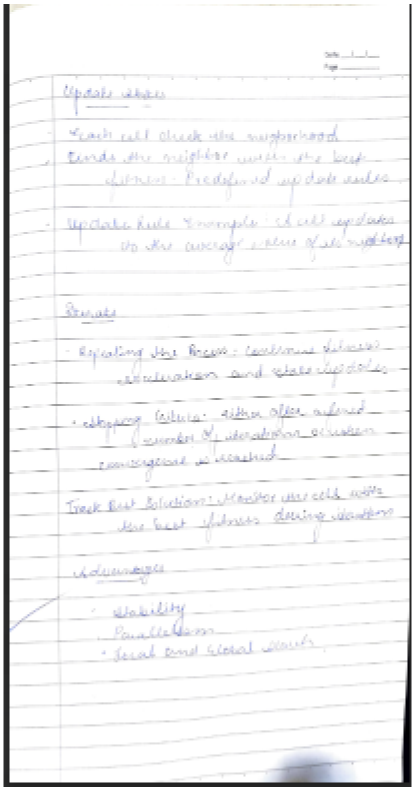
· Example: Each cell is randomly assigned a value from -10 to 10

Evaluate Fitness

· Fitness function: Measure the quality of each cell' & [illegible]

· example: $f(x) = x^2-4x+4$, fitness is calculated for each cell

· Cells with corresponding fitness values is measured.



Date: / /
Page:

Update Rules

· Each cell check the neighborhood
· finds the neighbor with the best fitness. Predefined update rules

· Update Rule Example: A cell updates to the average value of its neighbors

Iterate

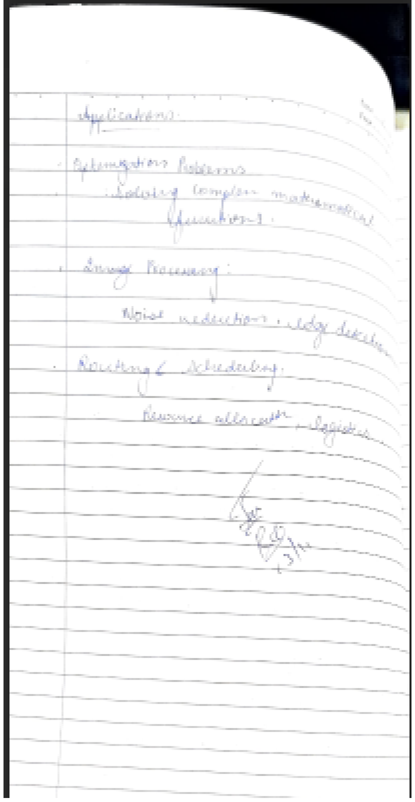
· Repeating the Process: continue fitness evaluation and state updates

· stopping criteria: either after a fixed number of iteration or when convergence is reached

Track Best Solution: Monitor the cells with the best fitness during iterations

Advantages

· Stability
· Parallelism
· local and global search



Applications:

· Optimization Problems
Solving complex mathematical functions.

· Image Processing:
Noise reduction, edge detection

· Routing & Scheduling.
Resource allocation, logistics

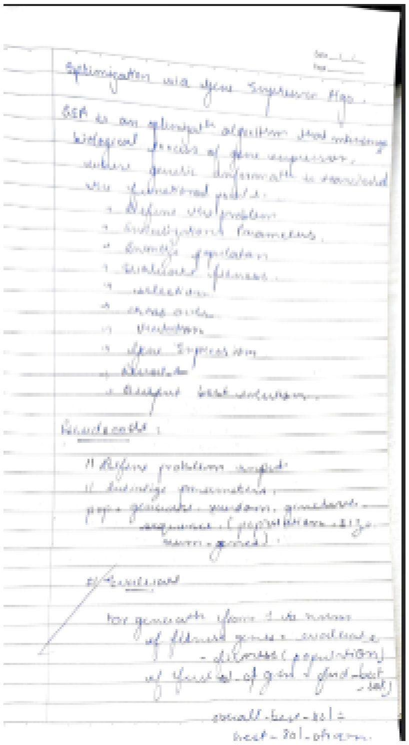
CODE:

```
import numpy as np

# Parameters
L = 30           # Road
length Vmax = 5
p_slow = 0.3
timesteps = 10
num_cars = 5

# Initialize road: -1 for empty, else speed of car
road = -1 * np.ones(L, dtype=int)
car_positions = np.random.choice(L, num_cars,
replace=False) road[car_positions] = 0

def update_road(road):
    new_road = -1 * np.ones_like(road)
    L = len(road)
    for i in range(L):
        if road[i] != -1: # There's a car here
            v = road[i]

            # Calculate gap to next car
            distance = 1
            while distance <= Vmax:
                check_pos = (i + distance) %
                L if road[check_pos] != -1:
                    break
                distance += 1
            gap = distance - 1

            # Step 1:
            Acceleration v =
            min(v + 1, Vmax)

            # Step 2: Slowing down due to cars
            v = min(v, gap)

            # Step 3: Random slow down
            if v > 0 and np.random.random() < p_slow:
                v -= 1

            # Step 4: Move car
            new_pos = (i + v) % L

            # Check for collisions (should not happen if rules are correct)
            if new_road[new_pos] == -1:
                new_road[new_pos] = v
            else:
                # If collision, keep old position (very
                unlikely) new_road[i] = v
    return new_road
def print_road(road):
    print("".join(['1' if x != -1 else '0' for x in road]))

print("Initial state:")
print_road(road)

for t in range(timesteps):
    road = update_road(road)
    print(f"Step {t + 1}:")
    print_road(road)
```

PROGRAM 7:

Optimization via Gene Expression Algorithms

ALGORITHM:



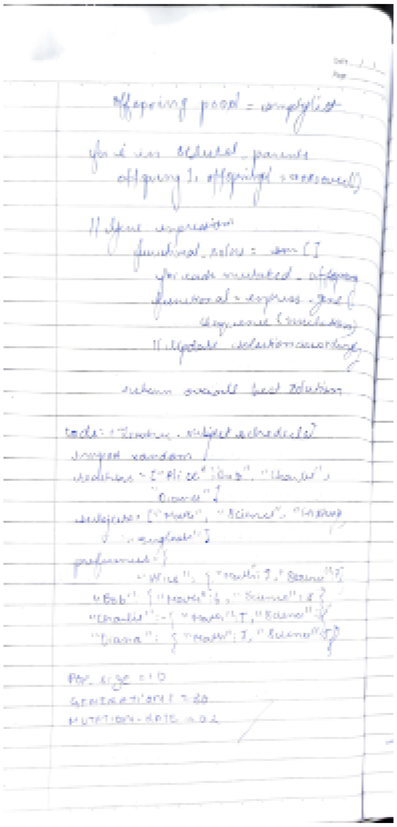
Optimization via Gene Expression Algo.

GEA is an optimization algorithm that mimics biological process of gene expression, where genetic information is translated into functional proteins.

- Define the problem
- Initialize Parameters.
- Initialize population
- Evaluate fitness.
- Selection
- Crossover
- Mutation
- Gene Expression
- Iterate
- Output best solution.

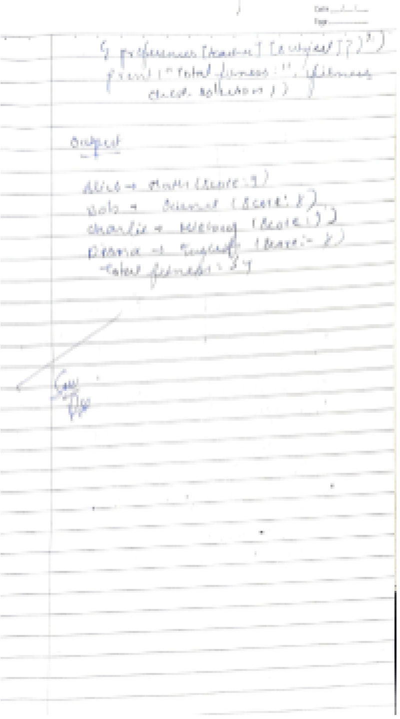
Pseudocode:

```
// Define problem input
// Initialize parameters.
pop = generate_random_gene_sequence(population_size,
        num_genes)

// Evolution
    for generation from 1 to num
        of [illegible] genes = evaluate
            _fitness(population)
        if [illegible] of gen > gen_best
                                   _sol)
            overall_best_sol =
              best_sol_of_gen.
```



```
        offspring_pool = emptylist
    for i in selected_parents
        offspring1, offspring2 = crossover()
    // gene expression
        functional_soln = []
            for each mutated_offspring
            functional = express_gene(
                sequence(mutated)
            // update solution accordingly
        return overall best solution
```

```
code: # Teacher_subject schedule
import random
teachers = ["Alice","Bob","Charlie",
        "Diana"]
subjects = ["Math", "Science", "History"
    , "English"]
preferences = {
        "Alice": {"Math":[illegible], "Science":[illegible]}
    "Bob": {"Math":[illegible], "Science":[illegible]}
    "Charlie": {"Math":[illegible], "Science":[illegible]}
    "Diana": {"Math":[illegible], "Science":[illegible]}

POP_SIZE = 10
GENERATIONS = 20
MUTATION_RATE = 0.1
```



```
{preferences[student][subject]}')
print("Total fitness:", fitness(
    best_solution))
```

Output

```
Alice → Math (Score: 9)
Bob → Science (Score: 8)
Charlie → History (Score: 9)
Diana → English (Score: 8)
Total fitness: 34
```

CODE:

```
import random
import math

# Cities (x, y coordinates)
cities = [(0,0), (1,3), (4,3), (6,1), (3,0)]

# Distance between two cities
def dist(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

# Total route distance
def route_distance(route):
    return sum(dist(cities[route[i]], cities[route[(i+1)%len(route)]]) for i in range(len(route)))

# Fitness (shorter distance = better)
def fitness(route):
    return 1 / route_distance(route)
```

```
# Create initial
population def
init_population(size):
    base = list(range(len(cities)))
    return [random.sample(base, len(base)) for _ in range(size)]

# Selection (roulette
wheel) def select(pop, fits):
    total = sum(fits)
    pick = random.uniform(0, total)
    curr = 0
    for i, f in
        enumerate(fits): curr
        += f
        if curr > pick:
            return pop[i]

# Crossover (ordered crossover)
def crossover(p1, p2):
    a, b = sorted(random.sample(range(len(p1)), 2))
    child = [None]*len(p1)
    child[a:b] = p1[a:b]
    ptr = 0
    for x in p2:
        if x not in child:
            while child[ptr] is not None:
                ptr += 1
            child[ptr] = x
    return child

# Mutation (swap two
cities) def mutate(route):
    i, j = random.sample(range(len(route)),
    2) route[i], route[j] = route[j], route[i]
    return route

# Main GA
loop POP_SIZE
= 4
GENERATIONS = 5
pop = init_population(POP_SIZE)

for gen in range(GENERATIONS):
    fits = [fitness(r) for r in pop]
    best_route = pop[fits.index(max(fits))]
    print(f"Gen {gen+1}: Best distance = {route_distance(best_route):.2f}, Route = {best_route}")

    new_pop = []
    for _ in range(POP_SIZE):
        p1, p2 = select(pop, fits), select(pop, fits)
```

```
child = crossover(p1, p2)
if random.random() < 0.2:
```

```
            child = mutate(child)
        new_pop.append(child)
    pop = new_pop

# Final Result
fits = [fitness(r) for r in pop]
best_route = pop[fits.index(max(fits))]
print("\nFinal Best Route:",
best_route)
print("Final Best Distance:", route_distance(best_route))
```