VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Samhitha A (1BM23CS293)

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 **Samhitha A (1BM23CS293),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

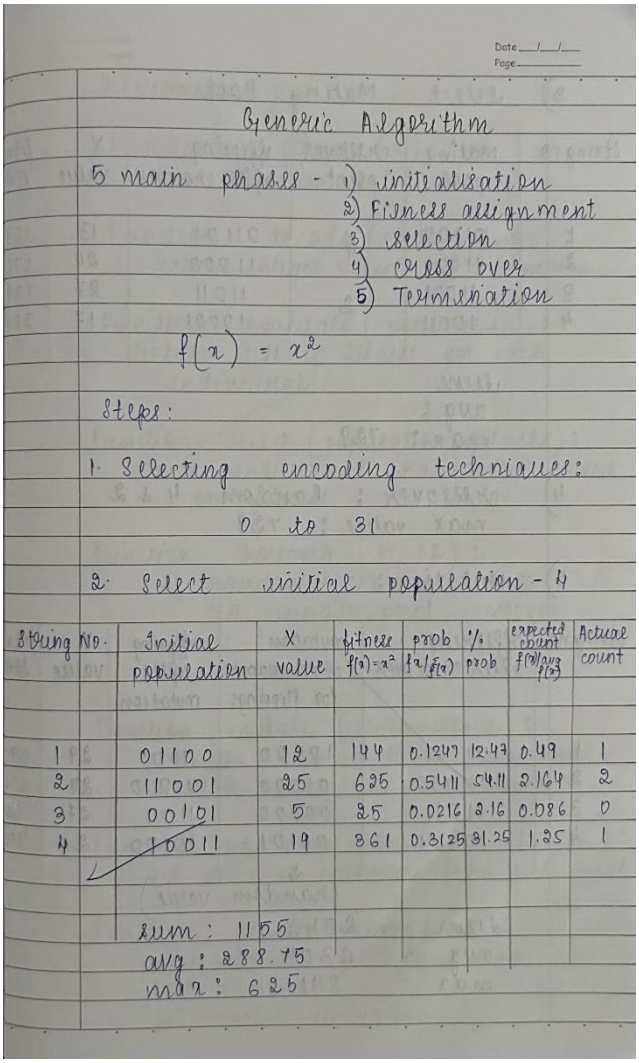
Index

Github Link:
https://github.com/Samhithagit/1BM23CS293_BIS_LAB

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:



Genetic Algorithm

5 main phases - 1) initialisation
2) Fitness alignment
3) selection
4) cross over
5) Termination

$f(x) = x^2$

Steps:

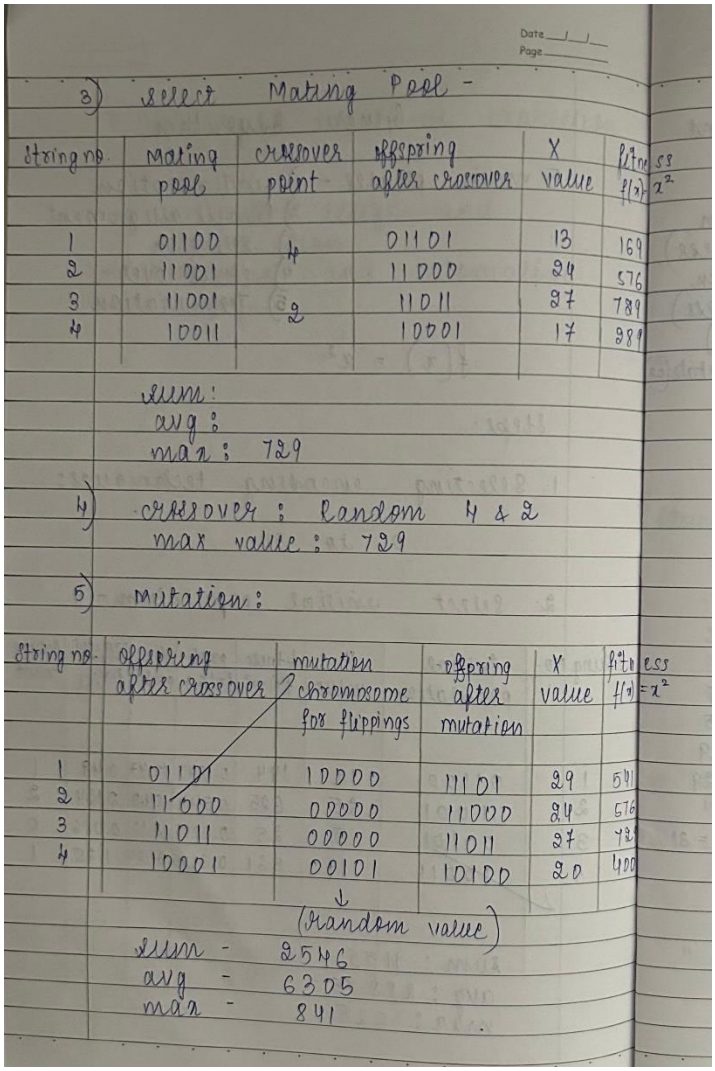
1. Selecting encoding techniques:

0 to 31

2. Select initial population - 4

String No.	Initial population	X value	fitness $f(x)=x^2$	prob $f_i/\sum f(x)$	% prob	expected count $f(x)/\text{avg}$	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

sum : 1155
avg : 288.75
max : 625



3) Select Mating Pool -

String no.	Mating pool	crossover point	offspring after crossover	X value	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	17	289

sum:
avg:
max: 729

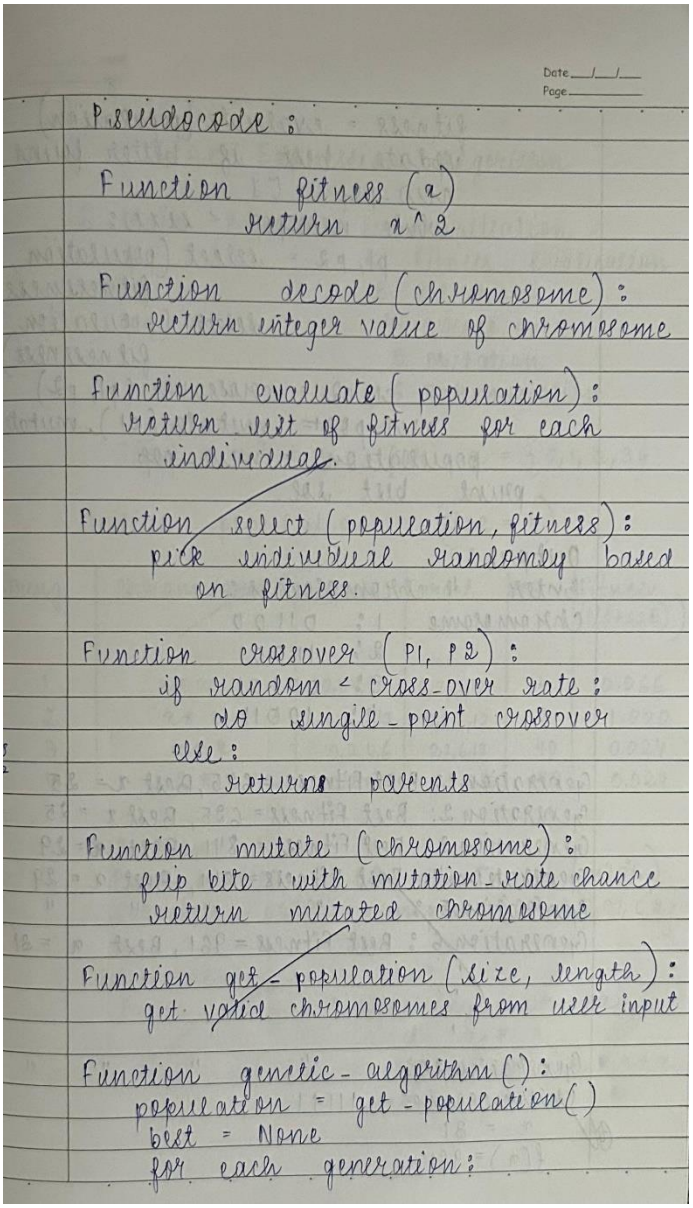
4) crossover : Random 4 & 2
max value : 729

5) Mutation:

String no.	offspring after crossover	mutation chromosome for flippings	offspring after mutation	X value	fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400

(Random value)

sum - 2546
avg - 6305
max - 841



Pseudocode :

Function fitness (x)
return x ^ 2

Function decode (chromosome):
return integer value of chromosome

Function evaluate (population):
return list of fitness for each individual

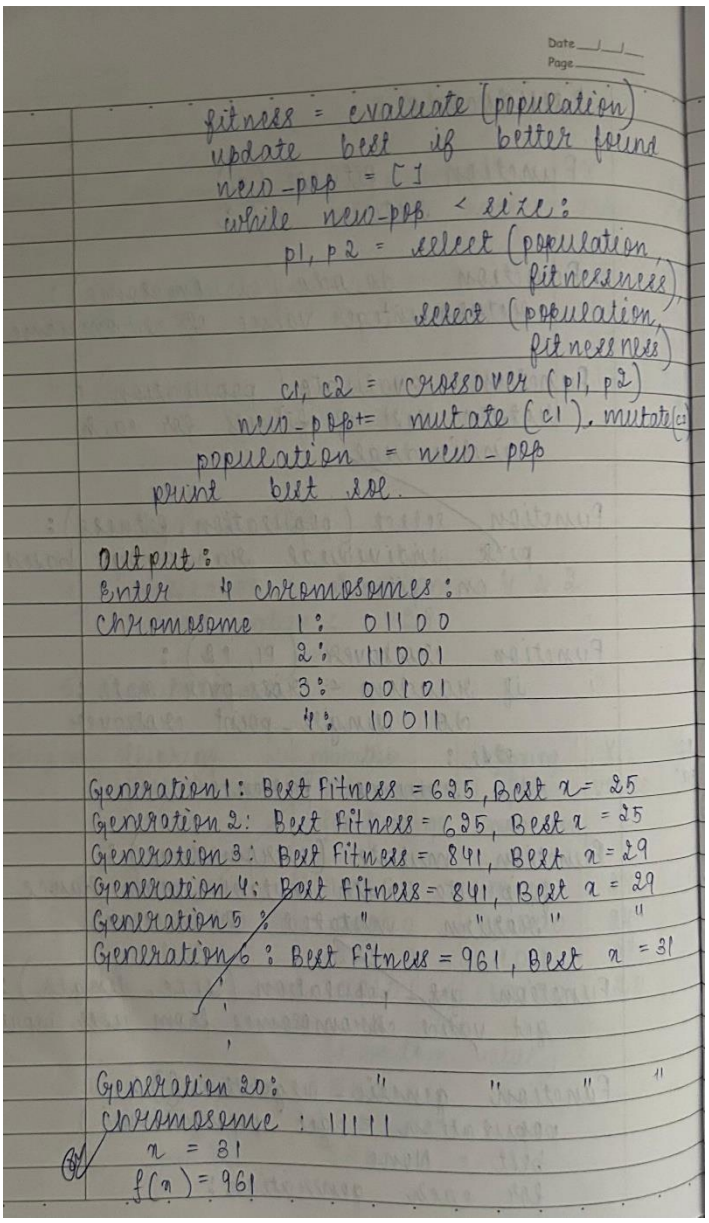
Function select (population, fitness):
pick individual randomly based on fitness.

Function crossover (P1, P2):
if random < cross-over rate:
do single-point crossover
else:
returns parents

Function mutate (chromosome):
flip bits with mutation-rate chance
return mutated chromosome

Function get-population (size, length):
get valid chromosomes from user input

Function genetic-algorithm():
population = get-population()
best = None
for each generation:



fitness = evaluate (population)
update best if better found
new-pop = []
while new-pop < size:
p1, p2 = select (population, fitnessness)
select (population, fitnessness)
c1, c2 = crossover (p1, p2)
new-pop += mutate (c1), mutate(c2)
population = new-pop
print best sol.

Output:
Enter 4 chromosomes:
Chromosome 1: 01100
2: 11001
3: 00101
4: 10011

Generation 1: Best Fitness = 625, Best x = 25
Generation 2: Best Fitness = 625, Best x = 25
Generation 3: Best Fitness = 841, Best x = 29
Generation 4: Best Fitness = 841, Best x = 29
Generation 5: " " " "
Generation 6: Best Fitness = 961, Best x = 31

Generation 20: " " " "
Chromosome : 11111
x = 31
f(x) = 961

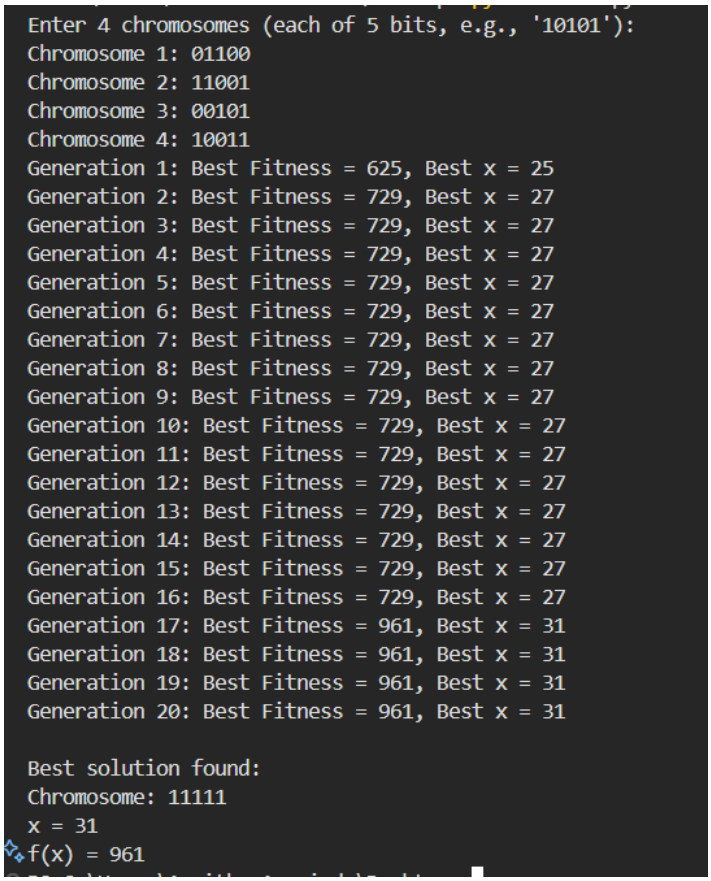
Code:

```
import random

def fitness_function(x):
    return x ** 2

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

def evaluate_population(population):
    return [fitness_function(decode(individual)) for individual in population]

def select(population, fitnesses):
```

```
    total_fitness = sum(fitnesses)
    if total_fitness == 0:
        return random.choice(population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, CHROMOSOME_LENGTH - 1)
        return (parent1[:point] + parent2[point:], parent2[:point] + parent1[point:])
    return parent1, parent2

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

def get_initial_population(size, length):
    population = []
    print(f"Enter {size} chromosomes (each of {length} bits, e.g., '10101'):")
    while len(population) < size:
        chrom = input(f"Chromosome {len(population)+1}: ").strip()
        if len(chrom) == length and all(bit in '01' for bit in chrom):
            population.append(chrom)
        else:
            print(f"Invalid input. Please enter a {length}-bit binary string.")
    return population

def genetic_algorithm():
    population = get_initial_population(POPULATION_SIZE, CHROMOSOME_LENGTH)
    best_solution = None
    best_fitness = float('-inf')

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)


        for i, individual in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
```

```
            best_solution = individual

        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best x =
 {decode(best_solution)}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

    print("\nBest solution found:")
    print(f"Chromosome: {best_solution}")
    print(f"x = {decode(best_solution)}")
    print(f"f(x) = {fitness_function(decode(best_solution))}")


 POPULATION_SIZE = 4
 CHROMOSOME_LENGTH = 5
 MUTATION_RATE = 0.01
 CROSSOVER_RATE = 0.8
 GENERATIONS = 20


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

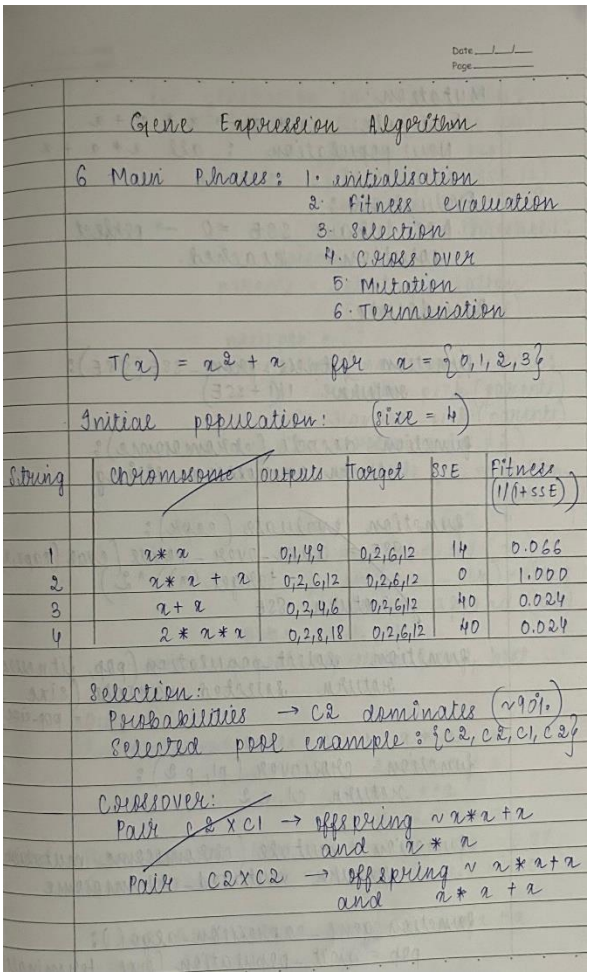
Output:


Enter 4 chromosomes (each of 5 bits, e.g., '10101'):
Chromosome 1: 01100
Chromosome 2: 11001
Chromosome 3: 00101
Chromosome 4: 10011
Generation 1: Best Fitness = 625, Best x = 25
Generation 2: Best Fitness = 729, Best x = 27
Generation 3: Best Fitness = 729, Best x = 27
Generation 4: Best Fitness = 729, Best x = 27
Generation 5: Best Fitness = 729, Best x = 27
Generation 6: Best Fitness = 729, Best x = 27
Generation 7: Best Fitness = 729, Best x = 27
Generation 8: Best Fitness = 729, Best x = 27
Generation 9: Best Fitness = 729, Best x = 27
Generation 10: Best Fitness = 729, Best x = 27
Generation 11: Best Fitness = 729, Best x = 27
Generation 12: Best Fitness = 729, Best x = 27
Generation 13: Best Fitness = 729, Best x = 27
Generation 14: Best Fitness = 729, Best x = 27
Generation 15: Best Fitness = 729, Best x = 27
Generation 16: Best Fitness = 729, Best x = 27
Generation 17: Best Fitness = 961, Best x = 31
Generation 18: Best Fitness = 961, Best x = 31
Generation 19: Best Fitness = 961, Best x = 31
Generation 20: Best Fitness = 961, Best x = 31
Best solution found:
Chromosome: 11111
x = 31
f(x) = 961

Program 2

Optimization via Gene Expression Algorithms:

Algorithm:



Gene Expression Algorithm

6 Main Phases: 1. initialisation
2. Fitness evaluation
3. Selection
4. Cross over
5. Mutation
6. Termination

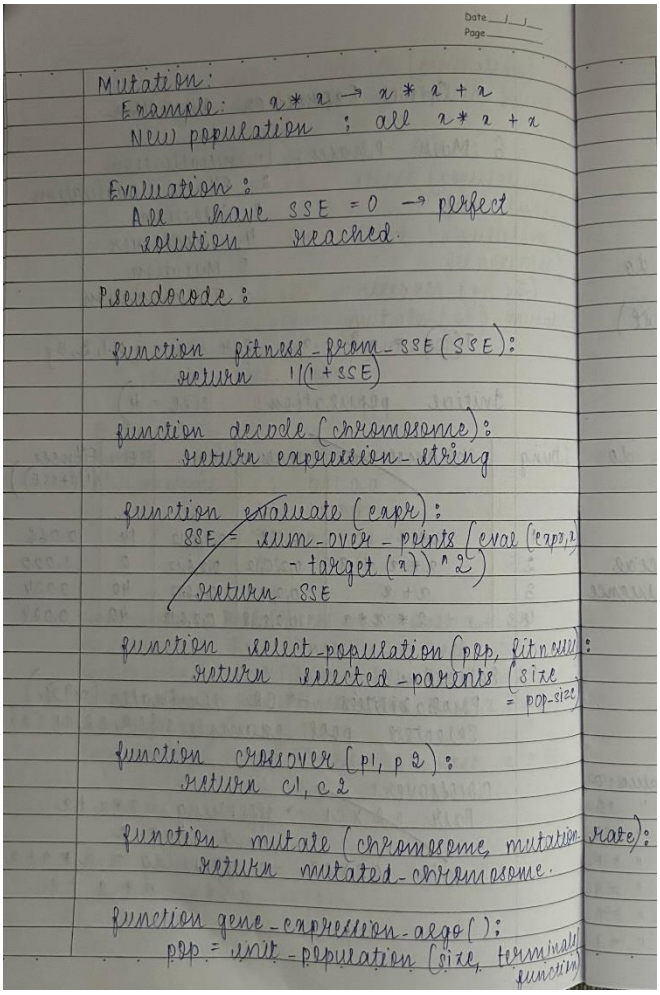
$T(x) = x^2 + x$ for $x = \{0, 1, 2, 3\}$

Initial population: (size = 4)

String	Chromosome	Expression	Target	SSE	Fitness (1/(1+SSE))
1	x * x	0,1,4,9	0,2,6,12	14	0.066
2	x * x + x	0,2,6,12	0,2,6,12	0	1.000
3	x + x	0,2,4,6	0,2,6,12	40	0.024
4	2 * x * x	0,2,8,18	0,2,6,12	40	0.024

Selection:
Probabilities → C2 dominates (~99%)
Selected pool example: {C2, C2, C1, C2}

Crossover:
Pair C2 x C1 → offspring ~ x*x+x and x * x
Pair C2 x C2 → offspring ~ x*x+x and x * x + x

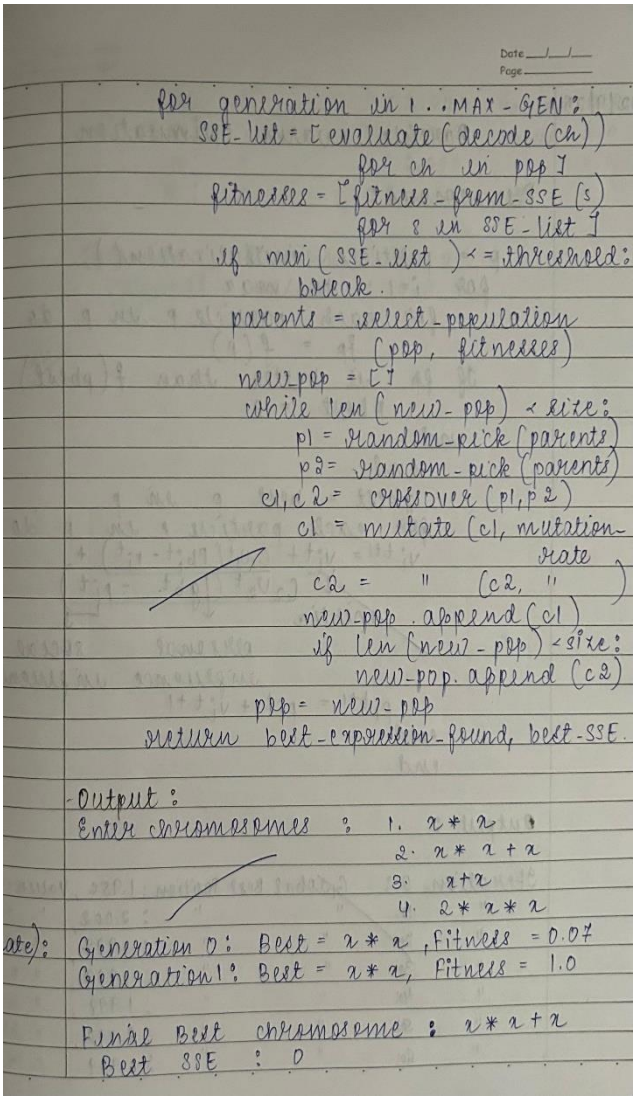


Mutation:
Example: x * x → x * x + x
New population : all x * x + x

Evaluation:
All have SSE = 0 → perfect solution reached.

Pseudocode:

```
function fitness_from_SSE(SSE):
    return 1/(1+SSE)

function decode(chromosome):
    return expression_string

function evaluate(expr):
    SSE = sum_over_points(eval(expr,x) - target(x))^2)
    return SSE

function select_population(pop, fitness):
    return selected_parents(size = pop_size)

function crossover(p1, p2):
    return c1, c2

function mutate(chromosome, mutation_rate):
    return mutated_chromosome

function gene_expression_algo():
    pop = init_population(size, terminals, functions)
```



Code:

```
import random
import math

def target_function(x):
    return x * math.sin(10 * math.pi * x) + 1.0


POPULATION_SIZE = 50
GENE_LENGTH = 10
GENERATIONS = 7
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
DOMAIN = (-1, 1)
FUNCTIONS = {
    "add": (lambda a, b: a + b, "+"),
```

```
    "sub": (lambda a, b: a - b, "-"),
    "mul": (lambda a, b: a * b, "*"),
    "div": (lambda a, b: a / b if b != 0 else 1, "/")
}
TERMINALS = ['x', 1.0, -1.0, 2.0]


def random_gene():
    gene = []
    for _ in range(GENE_LENGTH):
        if random.random() < 0.5:
            gene.append(random.choice(list(FUNCTIONS.keys())))
        else:
            gene.append(random.choice(TERMINALS))
    return gene

population = [random_gene() for _ in range(POPULATION_SIZE)]

def evaluate_gene(gene, x):
    stack = []
    for g in gene:
        if g in TERMINALS:
            stack.append(x if g == 'x' else g)
        elif g in FUNCTIONS:
            if len(stack) >= 2:
                b = stack.pop()
                a = stack.pop()
                stack.append(FUNCTIONS[g][0](a, b))
    return stack[0] if stack else 0


def fitness(gene):
    total_error = 0.0
    for _ in range(10):
        x = random.uniform(*DOMAIN)
        try:
            y_pred = evaluate_gene(gene, x)
        except:
            y_pred = 0
        y_true = target_function(x)
        total_error += abs(y_true - y_pred)
    return 1 / (1 + total_error)

def select(pop):
    k = 3
    candidates = random.sample(pop, k)
    return max(candidates, key=lambda g: fitness(g))
```

```
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENE_LENGTH - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1[:], parent2[:]

def mutate(gene):
    for i in range(GENE_LENGTH):
        if random.random() < MUTATION_RATE:
            if random.random() < 0.5:
                gene[i] = random.choice(list(FUNCTIONS.keys()))
            else:
                gene[i] = random.choice(TERMINALS)
    return gene

def gene_to_expression(gene):
    stack = []
    for g in gene:
        if g in TERMINALS:
            stack.append(str(g))
        elif g in FUNCTIONS:
            if len(stack) >= 2:
                b = stack.pop()
                a = stack.pop()
                op = FUNCTIONS[g][1]
                stack.append(f"({a} {op} {b})")
    return stack[0] if stack else "0"

best_gene = None
best_fit = -float("inf")

for gen in range(GENERATIONS):
    new_population = []
    while len(new_population) < POPULATION_SIZE:
        p1 = select(population)
        p2 = select(population)
        c1, c2 = crossover(p1, p2)
        c1 = mutate(c1)
        c2 = mutate(c2)
        new_population.extend([c1, c2])

    population = new_population[:POPULATION_SIZE]


    for gene in population:
        f = fitness(gene)
```

```
        if f > best_fit:
            best_fit = f
            best_gene = gene

    print(f"Generation {gen+1}: Best Fitness = {best_fit:.5f}")


print("\nBest Gene (raw):")
print(" ".join(map(str, best_gene)))
print("Best Gene (expression):")
print(gene_to_expression(best_gene))
print("Best Fitness:", best_fit)
```

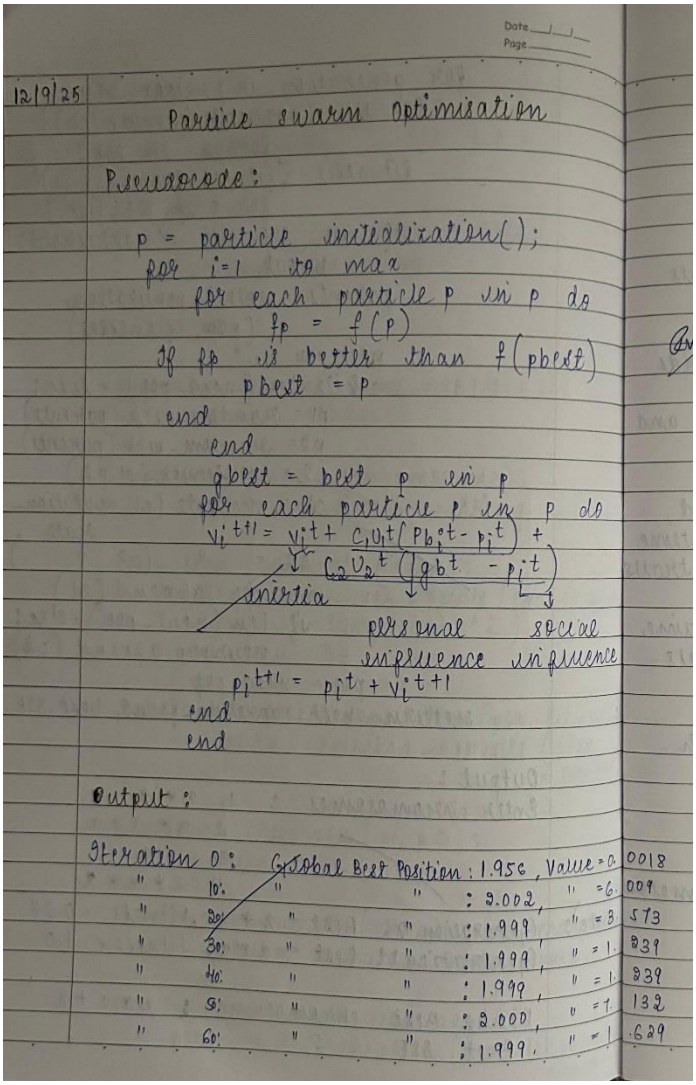
Output:

```
Generation 1: Best Fitness = 0.41053
Generation 2: Best Fitness = 0.41053
Generation 3: Best Fitness = 0.41053
Generation 4: Best Fitness = 0.47526
Generation 5: Best Fitness = 0.47526
Generation 6: Best Fitness = 0.47526
Generation 7: Best Fitness = 0.47526

Best Gene (raw):
mul mul mul -1.0 -1.0 mul div mul add -1.0
Best Gene (expression):
(-1.0 * -1.0)
Best Fitness: 0.47525613121342497
```

Program 3

Particle Swarm Optimization for Function Optimization:

Algorithm:



12/9/25

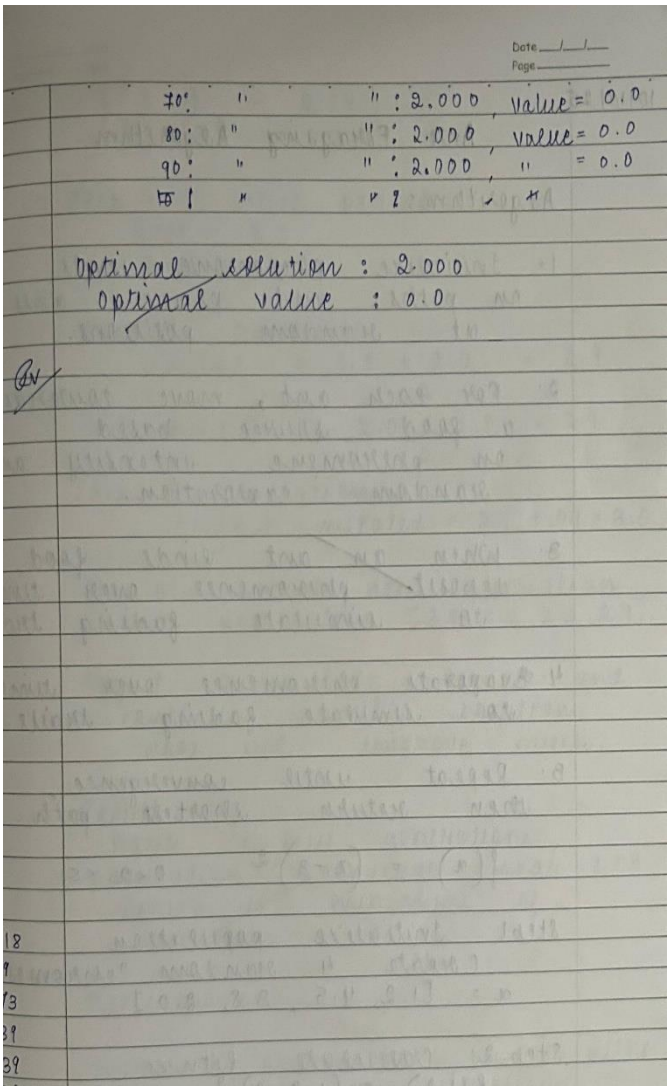
Particle swarm optimisation

Pseudocode:

p = particle initialization();
for i=1 to max
for each particle p in p do
fp = f(p)
if fp is better than f(pbest)
pbest = p
end
end
gbest = best p in p
for each particle p in p do
$v_i^{t+1} = v_i^t + c_1 U_1^t (P_{b_i}^t - p_i^t) + c_2 U_2^t (g_b^t - p_i^t)$
inertia — personal influence — social influence
$p_i^{t+1} = p_i^t + v_i^{t+1}$
end
end

Output:

Iteration 0: Global Best Position: 1.956, Value = 0.0018
" 10: " " : 2.002, " = 6.009
" 20: " " : 1.999, " = 3.573
" 30: " " : 1.999, " = 1.839
" 40: " " : 1.999, " = 1.839
" 50: " " : 2.000, " = 1.132
" 60: " " : 1.999, " = 1.629



70: " " : 2.000, value = 0.0
80: " " : 2.000, value = 0.0
90: " " : 2.000, " = 0.0
100: " " : 2

Optimal solution : 2.000
Optimal value : 0.0

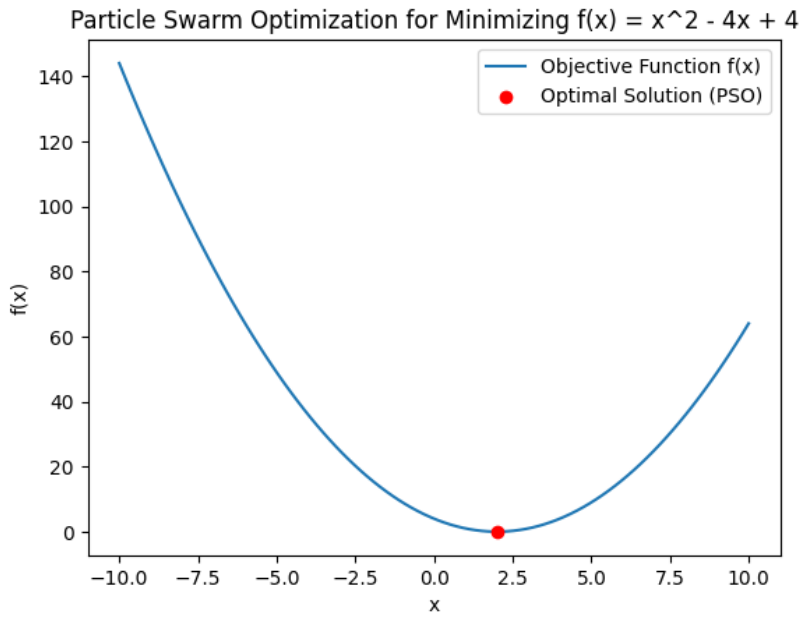
Code:

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


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

class Particle:
    def __init__(self, lower_bound, upper_bound):
        self.position = np.random.uniform(lower_bound, upper_bound)
        self.velocity = np.random.uniform(-1, 1)
        self.best_position = self.position
        self.best_value = objective_function(self.position)

    def update(self, global_best_position, w, c1, c2):
        r1 = np.random.rand()
        r2 = np.random.rand()


        self.velocity = w * self.velocity + c1 * r1 * (self.best_position - self.position) + c2 * r2 *
(global_best_position - self.position)
        self.position += self.velocity


        current_value = objective_function(self.position)


        if current_value < self.best_value:
            self.best_value = current_value
            self.best_position = self.position

# PSO parameters
num_particles = 30
num_iterations = 100
w = 0.7
c1 = 1.5
c2 = 1.5
lower_bound = -10
upper_bound = 10


particles = [Particle(lower_bound, upper_bound) for _ in range(num_particles)]
```

```
global_best_position = particles[0].best_position
global_best_value = particles[0].best_value

# PSO loop
for iteration in range(num_iterations):
    for particle in particles:
        particle.update(global_best_position, w, c1, c2)


        if particle.best_value < global_best_value:
            global_best_value = particle.best_value
            global_best_position = particle.best_position


    if iteration % 10 == 0:
        print(f"Iteration {iteration}: Global Best Position = {global_best_position}, Value =
{global_best_value}")


print(f"\nOptimal Position: {global_best_position}")
print(f"Optimal Value: {global_best_value}")


x_values = np.linspace(lower_bound, upper_bound, 100)
y_values = objective_function(x_values)

plt.plot(x_values, y_values, label="Objective Function f(x)")
plt.scatter(global_best_position, global_best_value, color='red', label='Optimal Solution (PSO)', zorder=5)
plt.title("Particle Swarm Optimization for Minimizing f(x) = x^2 - 4x + 4")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.show()
```

Output:

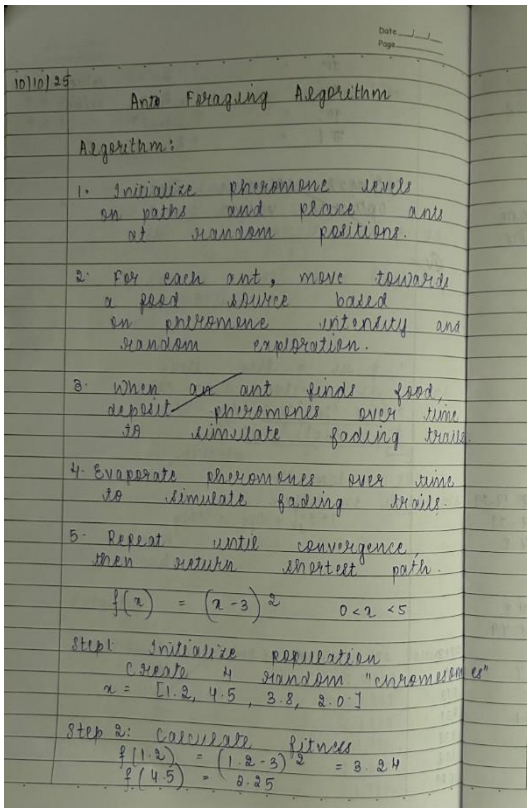
```
Iteration 0: Global Best Position = 1.9568424573953447, Value = 0.0018625734836725805
Iteration 10: Global Best Position = 2.0024697985436197, Value = 6.0999048461241046e-06
Iteration 20: Global Best Position = 1.9994022073823243, Value = 3.573560136693743e-07
Iteration 30: Global Best Position = 1.9999888661943581, Value = 1.2396172976991693e-10
Iteration 40: Global Best Position = 1.9999888661943581, Value = 1.2396172976991693e-10
Iteration 50: Global Best Position = 2.0000026705903675, Value = 7.132072710192006e-12
Iteration 60: Global Best Position = 1.9999995963039738, Value = 1.6298074001497298e-13
Iteration 70: Global Best Position = 2.0000000171682375, Value = 0.0
Iteration 80: Global Best Position = 2.0000000171682375, Value = 0.0
Iteration 90: Global Best Position = 2.0000000171682375, Value = 0.0

Optimal Position: 2.0000000171682375
Optimal Value: 0.0
```



Program 4

Ant Colony Optimization for the Traveling Salesman Problem:

Algorithm:


10/10/25

Ant Foraging Algorithm

Algorithm:

1. Initialize pheromone levels on paths and place ants at random positions.

2. For each ant, move towards a food source based on pheromone intensity and random exploration.

3. When an ant finds food, deposit pheromones over time to simulate fading trails.

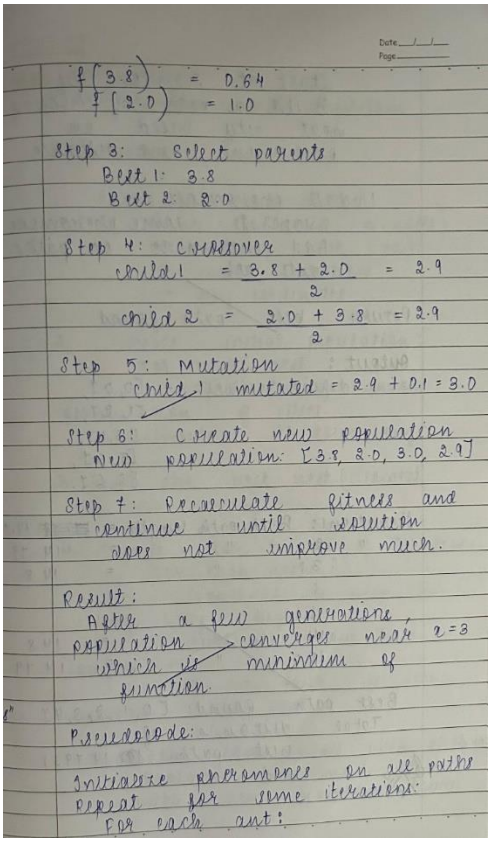
4. Evaporate pheromones over time to simulate fading trails.

5. Repeat until convergence, then return shortest path.

$f(x) = (x-3)^2 \quad 0 < x < 5$

Step 1: Initialize population
Create 4 random "chromosomes"
$x = [1.2, 4.5, 3.8, 2.0]$

Step 2: Calculate fitness
$f(1.2) = (1.2-3)^2 = 3.24$
$f(4.5) = 2.25$


$f(3.8) = 0.64$
$f(2.0) = 1.0$

Step 3: Select parents
Best 1: 3.8
Best 2: 2.0

Step 4: Crossover
child 1 $= \frac{3.8 + 2.0}{2} = 2.9$
child 2 $= \frac{2.0 + 3.8}{2} = 2.9$

Step 5: Mutation
child 1 mutated $= 2.9 + 0.1 = 3.0$

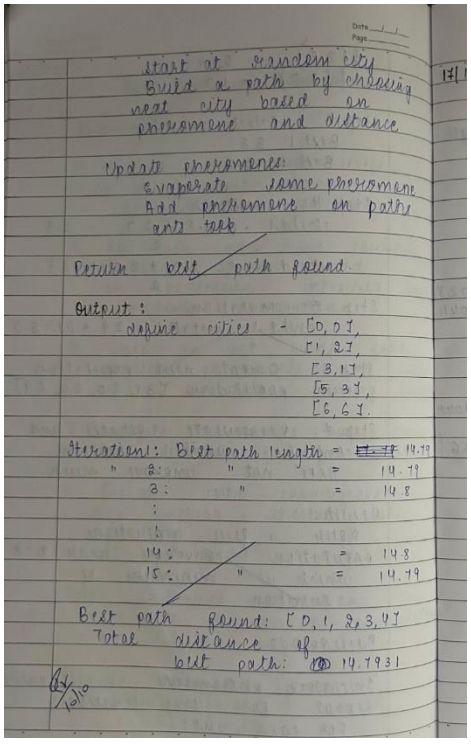
Step 6: Create new population
New population: [3.8, 2.0, 3.0, 2.9]

Step 7: Recalculate fitness and continue until solution does not improve much.

Result:
After a few generations, population converges near $x = 3$ which is minimum of function.

Pseudocode:

Initialize pheromones on all paths
Repeat for some iterations:
For each ant:


start at random city
Build a path by choosing next city based on pheromone and distance

Update pheromones:
Evaporate some pheromone
Add pheromone on paths ants took

Return best path found

Output:
Define cities = [0, 0],
[1, 2],
[3, 1],
[5, 3],
[6, 1].

Iterations: Best path length = ~~14.77~~ 14.79
2: " = 14.79
3: " = 14.8
:
:
14: " = 14.8
15: " = 14.79

Best path found: [0, 1, 2, 3, 4]
Total distance of best path: 14.7931

Code:

```
import numpy as np
import random

cities = np.array([
    [0, 0],
    [1, 2],
    [3, 1],
    [5, 3],
    [6, 6],
])

print("define cities -")
for city in cities:
    print(f"[{city[0]},{city[1]}],")
print()

class AntColony:
    def __init__(self, cities, n_ants, n_iterations, alpha=1, beta=2, rho=0.1, q=100):
        self.cities = cities
        self.n_cities = len(cities)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
```

```
    self.beta = beta
    self.rho = rho
    self.q = q
    self.pheromone = np.ones((self.n_cities, self.n_cities))
    self.distances = self.calculate_distances()

def calculate_distances(self):
    distances = np.zeros((self.n_cities, self.n_cities))
    for i in range(self.n_cities):
        for j in range(i + 1, self.n_cities):
            dist = np.linalg.norm(self.cities[i] - self.cities[j])
            distances[i][j] = dist
            distances[j][i] = dist
    return distances

def select_next_city(self, current_city, visited_cities):
    probabilities = []
    for next_city in range(self.n_cities):
        if next_city not in visited_cities:
            pheromone = self.pheromone[current_city][next_city] ** self.alpha
            distance = self.distances[current_city][next_city] ** -self.beta
            probabilities.append(pheromone * distance)
        else:
            probabilities.append(0)
```

```
    total_prob = sum(probabilities)
    probabilities = [p / total_prob for p in probabilities]
    return np.random.choice(range(self.n_cities), p=probabilities)

def update_pheromone(self, ant_paths, ant_lengths):
    self.pheromone *= (1 - self.rho)
    for i, path in enumerate(ant_paths):
        for j in range(len(path) - 1):
            self.pheromone[path[j]][path[j + 1]] += self.q / ant_lengths[i]
        self.pheromone[path[-1]][path[0]] += self.q / ant_lengths[i]

def optimize(self):
    shortest_path = None
    shortest_length = float('inf')

    for iteration in range(1, self.n_iterations + 1):
        ant_paths = []
        ant_lengths = []

        for _ in range(self.n_ants):
            path = [random.randint(0, self.n_cities - 1)]
            visited_cities = set(path)
            for _ in range(self.n_cities - 1):
                current_city = path[-1]
```

```
                next_city = self.select_next_city(current_city, visited_cities)
                path.append(next_city)
                visited_cities.add(next_city)
            path_length = sum(self.distances[path[i], path[i + 1]] for i in range(self.n_cities - 1))
            path_length += self.distances[path[-1], path[0]]
            ant_paths.append(path)
            ant_lengths.append(path_length)
            if path_length < shortest_length:
                shortest_length = path_length
                shortest_path = path
        self.update_pheromone(ant_paths, ant_lengths)
        print(f"Iteration {iteration} : Best path length = {shortest_length:.4f}")

    return shortest_path, shortest_length


n_ants = 10
n_iterations = 20
alpha = 1
beta = 2
rho = 0.1
q = 100

aco = AntColony(cities, n_ants, n_iterations, alpha, beta, rho, q)
```

```
    best_path, best_length = aco.optimize()


    print(f"\nBest path found : {best_path}")
print(f"Total distance of best path : {best_length:.4f}")
```

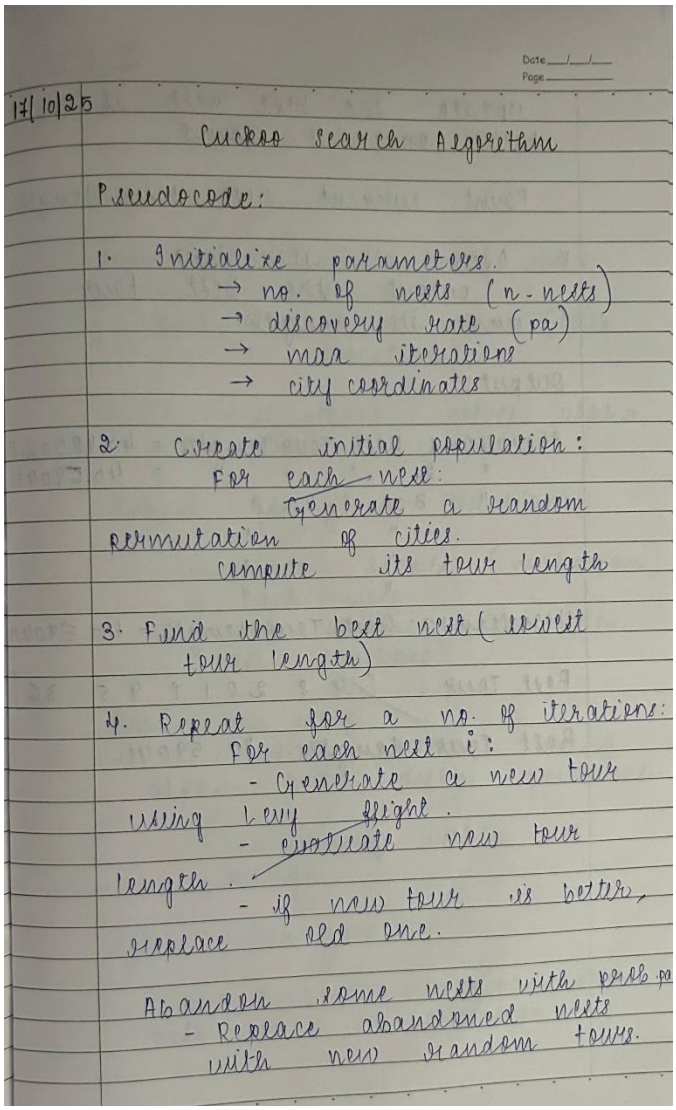
Output:

```
define cities -
[0,0],
[1,2],
[3,1],
[5,3],
[6,6],

Iteration 1 : Best path length = 17.7922
Iteration 2 : Best path length = 17.7922
Iteration 3 : Best path length = 17.7922
Iteration 4 : Best path length = 17.7922
Iteration 5 : Best path length = 17.7922
Iteration 6 : Best path length = 17.7922
Iteration 7 : Best path length = 17.7922
Iteration 8 : Best path length = 17.7922
Iteration 9 : Best path length = 17.7922
Iteration 10 : Best path length = 17.7922
Iteration 11 : Best path length = 17.7922
Iteration 12 : Best path length = 17.7922
Iteration 13 : Best path length = 17.7922
Iteration 14 : Best path length = 17.7922
Iteration 15 : Best path length = 17.7922
Iteration 16 : Best path length = 17.7922
Iteration 17 : Best path length = 17.7922
Iteration 18 : Best path length = 17.7922
Iteration 19 : Best path length = 17.7922
Iteration 20 : Best path length = 17.7922

Best path found : [0, np.int64(1), np.int64(4), np.int64(3), np.int64(2)]
Total distance of best path : 17.7922
```

Program 5

Cuckoo Search (CS)

Algorithm:



17/10/25

Cuckoo search Algorithm

Pseudocode:

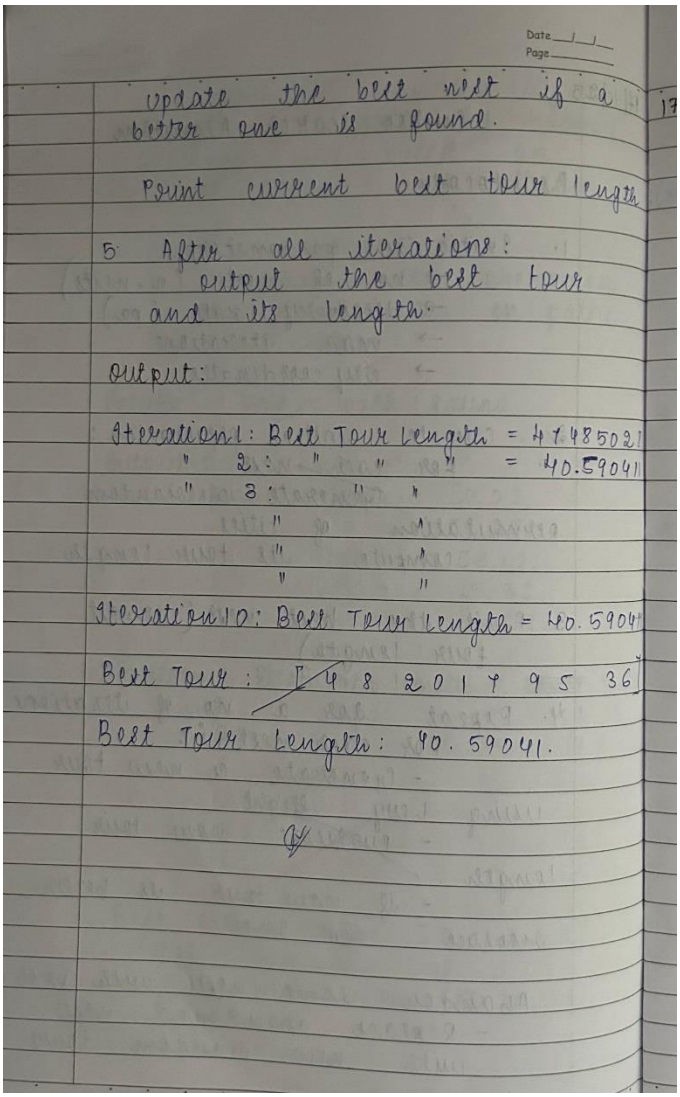
1. Initialize parameters.
   → no. of nests (n_nests)
   → discovery rate (pa)
   → max iterations
   → city coordinates

2. Create initial population:
   For each nest:
   Generate a random permutation of cities.
   Compute its tour length

3. Find the best nest (lowest tour length)

4. Repeat for a no. of iterations:
   For each nest i:
   - Generate a new tour using Levy flight.
   - Evaluate new tour length.
   - if new tour is better, replace old one.

Abandon some nests with prob pa
- Replace abandoned nests with new random tours.



update the best nest if a better one is found.

Print current best tour length

5. After all iterations:
   output the best tour and its length.

Output:

Iteration 1: Best Tour Length = 47.485021
" 2: " " " = 40.59041
" 3: " "
"
"
"
Iteration 10: Best Tour length = 40.59041

Best Tour: 4 8 2 0 1 7 9 5 3 6

Best Tour Length: 40.59041.

Code:

```
import numpy as np
import math
import random

def tour_length(tour, coords):
    tour_coords = coords[tour]
    d = np.sqrt(((tour_coords - np.roll(tour_coords, -1, axis=0))**2).sum(axis=1)).sum()
    return d

def levy_flight(Lambda, size):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2))) ** (1 / Lambda)
    sigma_v = 1.0
    u = np.random.normal(0, sigma_u, size)
    v = np.random.normal(0, sigma_v, size)
    step = u / (np.abs(v) ** (1 / Lambda))
    return step

def apply_levy_move(tour, coords, Lambda=1.5):
    n = len(tour)
    step = levy_flight(Lambda, n)
    magnitude = max(1, int(np.ceil(np.mean(np.abs(step)) * n / 4.0)))
    new_tour = tour.copy()
    for _ in range(magnitude):
        i, j = np.random.randint(0, n), np.random.randint(0, n)
        if i > j:
            i, j = j, i
        if i != j:
            new_tour[i:j+1] = new_tour[i:j+1][::-1]
    return new_tour

def cuckoo_search_tsp(coords, n_nests=20, pa=0.25, iterations=10):
    num_cities = len(coords)
    nests = np.array([np.random.permutation(num_cities) for _ in range(n_nests)])
    fitness = np.array([tour_length(t, coords) for t in nests])
    best_idx = np.argmin(fitness)
    best = nests[best_idx].copy()
    best_fitness = fitness[best_idx]
    for it in range(iterations):
        for i in range(n_nests):
            new_tour = apply_levy_move(nests[i], coords)
            new_fit = tour_length(new_tour, coords)
            if new_fit < fitness[i]:
                nests[i] = new_tour
                fitness[i] = new_fit
```

```
        abandon_mask = np.random.rand(n_nests) < pa
        for i in np.where(abandon_mask)[0]:
            nests[i] = np.random.permutation(num_cities)
            fitness[i] = tour_length(nests[i], coords)
        best_idx = np.argmin(fitness)
        if fitness[best_idx] < best_fitness:
            best_fitness = fitness[best_idx]
            best = nests[best_idx].copy()
        print(f"Iteration {it+1}: Best Tour Length = {best_fitness:.6f}")
    return best, best_fitness

if __name__ == "__main__":
    coords = np.array([
        [0,0],[1,5],[5,2],[6,6],[8,3],
        [2,9],[7,9],[3,4],[9,0],[4,7]
    ], dtype=float)
    best_tour, best_len = cuckoo_search_tsp(coords, n_nests=30, pa=0.25, iterations=10)
    print("\nBest tour (city indices):", best_tour)
    print("Best tour length:", best_len)
```

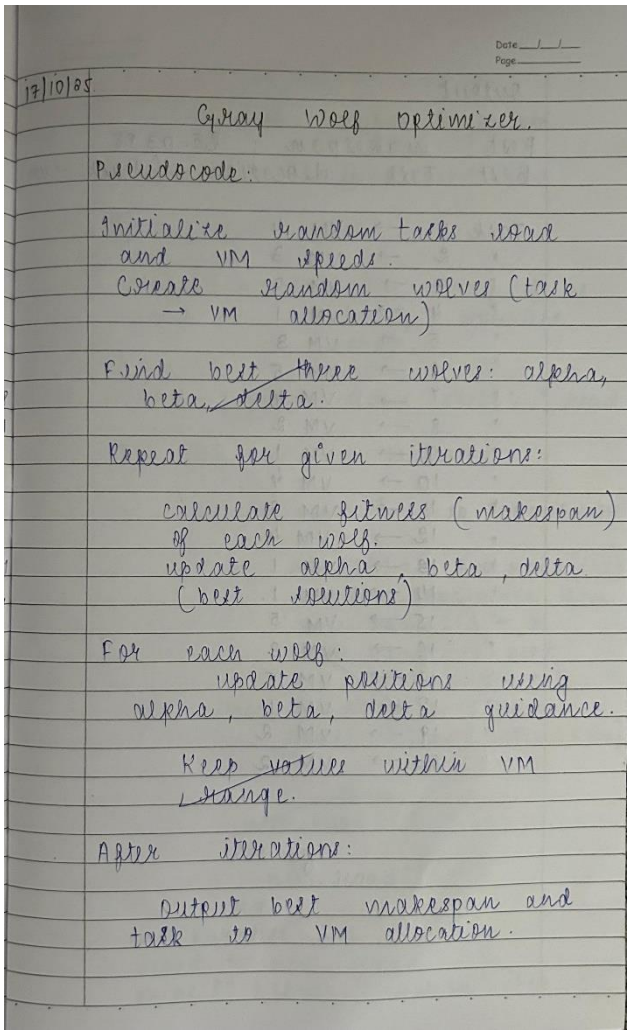
Output:

```
Iteration 1: Best Tour Length = 47.485028
Iteration 2: Best Tour Length = 40.590411
Iteration 3: Best Tour Length = 40.590411
Iteration 4: Best Tour Length = 40.590411
Iteration 5: Best Tour Length = 40.590411
Iteration 6: Best Tour Length = 40.590411
Iteration 7: Best Tour Length = 40.590411
Iteration 8: Best Tour Length = 40.590411
Iteration 9: Best Tour Length = 40.590411
Iteration 10: Best Tour Length = 40.590411

Best tour (city indices): [4 8 2 0 1 7 9 5 3 6]
Best tour length: 40.590410888776205
```

Program 6

Grey Wolf Optimizer (GWO):

Algorithm:


17/10/25

Gray Wolf Optimizer.

Pseudocode:

Initialize random tasks, load and VM speeds.
Create random wolves (task → VM allocation)

Find best three wolves: alpha, beta, delta.

Repeat for given iterations:

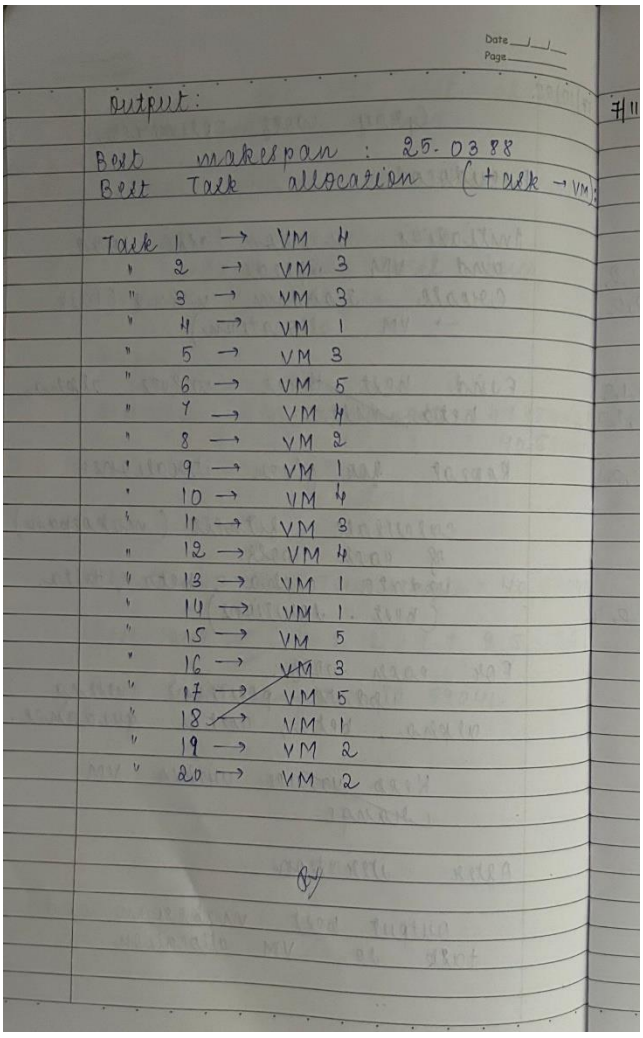
calculate fitness (makespan) of each wolf.
update alpha, beta, delta (best solutions)

For each wolf:
update positions using alpha, beta, delta guidance.

Keep values within VM range.

After iterations:

Output best makespan and task to VM allocation.


Output:

Best makespan : 25.0388
Best Task allocation (Task → VM):

Task 1 → VM 4
" 2 → VM 3
" 3 → VM 3
" 4 → VM 1
" 5 → VM 3
" 6 → VM 5
" 7 → VM 4
" 8 → VM 2
" 9 → VM 1
" 10 → VM 4
" 11 → VM 3
" 12 → VM 4
" 13 → VM 1
" 14 → VM 1
" 15 → VM 5
" 16 → VM 3
" 17 → VM 5
" 18 → VM 1
" 19 → VM 2
" 20 → VM 2

Code:

```
import numpy as np

num_tasks = 20
num_vms = 5
num_wolves = 15
max_iter = 50
```

```
task_load = np.random.randint(1000, 10000, num_tasks)
vm_speed = np.random.randint(500, 2000, num_vms)

def fitness(position):
    loads = np.zeros(num_vms)
    for i, vm in enumerate(position.astype(int)):
        loads[vm] += task_load[i] / vm_speed[vm]
    return np.max(loads)

wolves = np.random.randint(0, num_vms, (num_wolves, num_tasks))
alpha, beta, delta = None, None, None
alpha_score, beta_score, delta_score = np.inf, np.inf, np.inf

for t in range(max_iter):
    a = 2 - 2 * t / max_iter
    for i in range(num_wolves):
        score = fitness(wolves[i])
        if score < alpha_score:
            alpha_score, alpha = score, wolves[i].copy()
        elif score < beta_score:
            beta_score, beta = score, wolves[i].copy()
        elif score < delta_score:
            delta_score, delta = score, wolves[i].copy()

    for i in range(num_wolves):
        for j in range(num_tasks):
            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

            new_pos = (X1 + X2 + X3) / 3
            wolves[i][j] = np.clip(round(new_pos), 0, num_vms - 1)

best_allocation = alpha.astype(int)
best_makespan = alpha_score
```

```
print("Best Makespan:", best_makespan)
print("Best Task Allocation (task → VM):")
for i, vm in enumerate(best_allocation):
    print(f" Task {i+1} → VM {vm+1}")
```

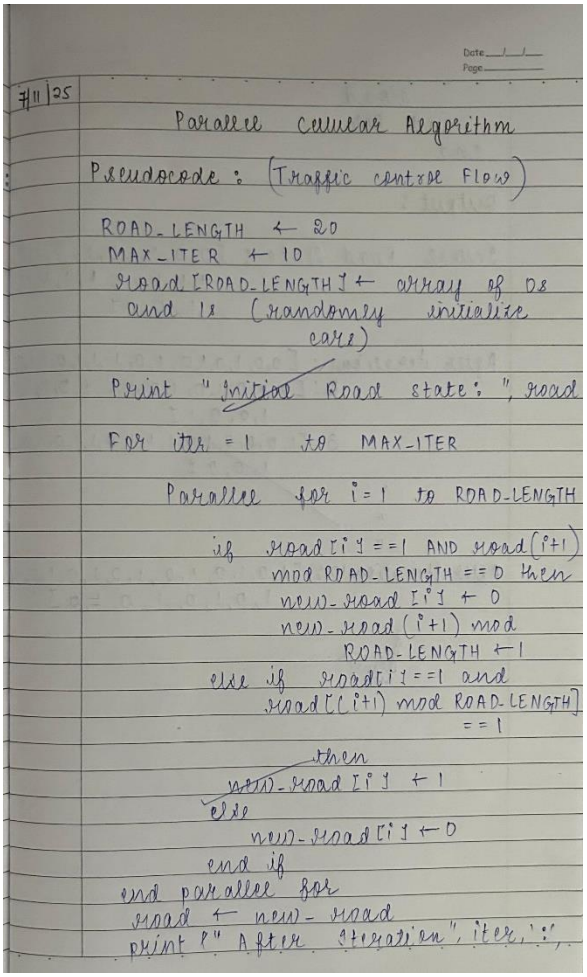
Output:

```
Best Makespan: 25.03882279273638
Best Task Allocation (task → VM):
 Task 1 → VM 4
 Task 2 → VM 3
 Task 3 → VM 3
 Task 4 → VM 1
 Task 5 → VM 3
 Task 6 → VM 5
 Task 7 → VM 4
 Task 8 → VM 2
 Task 9 → VM 1
 Task 10 → VM 4
 Task 11 → VM 3
 Task 12 → VM 4
 Task 13 → VM 1
 Task 14 → VM 1
 Task 15 → VM 5
 Task 16 → VM 3
 Task 17 → VM 5
 Task 18 → VM 1
 Task 19 → VM 2
 Task 20 → VM 2
```

Program 7

Parallel Cellular Algorithms and Programs:

Algorithm:


Parallel cellular Algorithm

Pseudocode : (Traffic control Flow)

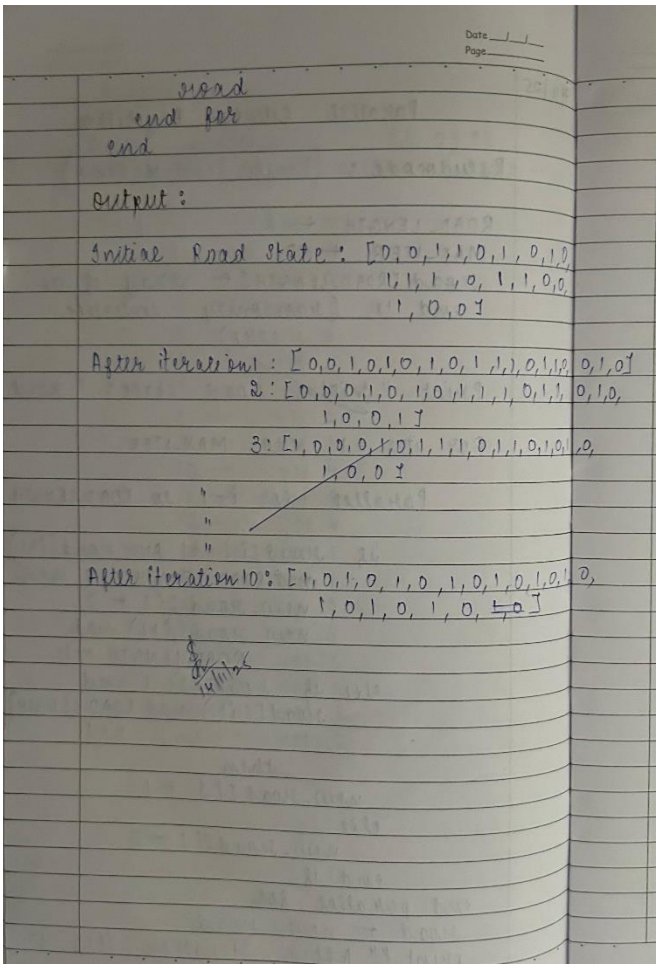
ROAD_LENGTH ← 20
MAX_ITER ← 10
road [ROAD_LENGTH] ← array of 0s and 1s (randomly initialize cars)

Print "Initial Road state: ", road

For iter = 1 to MAX_ITER

Parallel for i = 1 to ROAD_LENGTH

if road[i] == 1 AND road[(i+1) mod ROAD_LENGTH == 0 then
new_road[i] ← 0
new_road[(i+1) mod ROAD_LENGTH] ← 1
else if road[i] == 1 and road[(i+1) mod ROAD_LENGTH] == 1
then
new_road[i] ← 1
else
new_road[i] ← 0
end if
end parallel for
road ← new_road
print "After iteration", iter, ":",


road
end for
end

Output :

Initial road state : [0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0]

After iteration 1 : [0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0]
2 : [0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0]
3 : [1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0]
"
"
After iteration 10 : [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0]

Code:

```
import random

ROAD_LENGTH = 20
MAX_ITER = 10

road = [random.choice([0, 1]) for _ in
range(ROAD_LENGTH)]
print("Initial Road State:", road)

for iter in range(1, MAX_ITER + 1):
new_road = [0] * ROAD_LENGTH
```

```
for i in range(ROAD_LENGTH):
next_cell = (i + 1) % ROAD_LENGTH

if road[i] == 1 and road[next_cell] == 0:
new_road[next_cell] = 1 # Car moves forward
elif road[i] == 1 and road[next_cell] == 1:
new_road[i] = 1 # Car stays (blocked)
else:
new_road[i] = new_road[i] or 0 # Empty
remains empty
road = new_road.copy()

print(f"After iteration {iter}:", road)
```

Output: