VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Rachit Chandra (1BM23CS255)

in partial fulfilment 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
Sep-2024 to Jan-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 **Rachit Chandra (1BM23CS255),** 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.

Ms. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Rachit2510/BIS

Program 1

Genetic Algorithm for Optimization

Algorithm:

Date 28-2-25
Page (5)

Lab -2, Genetic Algorithm

- Algo

① Initialization

→ Define population size, crossover rate, mutation rate, max generation

→ Create an initial population of random individuals

② Evaluate

For each individual in population compute fitness using fitness function.

③ Loops (for each generation):

a) Selection :-

Choose parents based on fitness

b) Crossover :-

With probability crossover-rate, combine genes of two parents.

c) Mutation :-

With probability mutation rate, randomly change some genes.

d) Evaluation :-

Compute fitness for all new individuals

e) Replacement :-

From new population

- Pseudocode

Genetic Algo()

Initialize population P with N random chromosomes

Evaluate fitness of each chromosome in P

repeat

Date
Page (5)

select parent chromosomes from P based on fitness
with probability P_c, crossover selected
parents to generate offspring
with probability P_m, mutation some offspring
genes
Evaluate fitness of new offspring
from new population P' using offspring &
some best individuals from P
set P = P'
till termination condition is met
return the best chromosome found in P.

Code:

```
# Genetic Algorithm for single-variable optimization (real-coded)
# This code runs a GA to minimize a simple test function and prints one "best" per generation
# in the same textual format as the notebook image.
# You can change the function `f(x)` to any other objective you want to optimize.

import random
import math
import statistics

random.seed(42)

# Objective function to minimize
def f(x):
    # Example: simple convex bowl with minimum near x = 1.05 and base value 2.0
    return 2.0 + (x - 1.05)**2 * 0.9  # scale factor to make numbers similar to the image

# GA parameters
POP_SIZE = 20
GENS = 10
X_MIN, X_MAX = -2.0, 4.0
TOURNAMENT_SIZE = 3
CROSSOVER_PROB = 0.8
MUTATION_STD = 0.05  # gaussian mutation standard deviation
ELITISM = 1          # number of elites to carry to next generation

# Initialize population (real-coded)
population = [random.uniform(X_MIN, X_MAX) for _ in range(POP_SIZE)]

def tournament_select(pop, k=TOURNAMENT_SIZE):
    """Return one selected individual (real value) by tournament."""
    aspirants = random.sample(pop, k)
    aspirants.sort(key=lambda ind: f(ind))  # minimize
    return aspirants[0]

def blend_crossover(a, b, alpha=0.5):
    """Simple blend crossover (arithmetic mix)"""
    r = random.random()
    child = a * r + b * (1-r)
    return child

def mutate(x, sigma=MUTATION_STD):
    """Gaussian mutation (keeps within bounds)"""
```

```python
    x2 = x + random.gauss(0, sigma)
    # clip to bounds
    if x2 < X_MIN: x2 = X_MIN
    if x2 > X_MAX: x2 = X_MAX
    return x2

# Run GA and print best per generation
print("Optimization log:")
for gen in range(1, GENS+1):
    # Evaluate and keep elite(s)
    population.sort(key=lambda ind: f(ind))
    elites = population[:ELITISM]

    # Logging best of current population (before reproduction)
    best = population[0]
    best_val = f(best)
    print(f"Generation {gen}: Best x = {best:.5f} , f(x) = {best_val:.6f}")

    # Create new population
    new_pop = elites.copy()
    while len(new_pop) < POP_SIZE:
        # selection
        p1 = tournament_select(population)
        p2 = tournament_select(population)
        # crossover
        if random.random() < CROSSOVER_PROB:
            c = blend_crossover(p1, p2)
        else:
            c = p1
        # mutation
        c = mutate(c)
        new_pop.append(c)

    population = new_pop

# Final best
population.sort(key=lambda ind: f(ind))
best = population[0]
print("\nFinal best:")
print(f"Best x = {best:.6f} , f(x) = {f(best):.6f}")
```

```
Optimization log:
Generation 1: Best x = 1.03213 , f(x) = 2.000287
Generation 2: Best x = 1.03213 , f(x) = 2.000287
Generation 3: Best x = 1.04859 , f(x) = 2.000002
Generation 4: Best x = 1.04859 , f(x) = 2.000002
Generation 5: Best x = 1.05108 , f(x) = 2.000001
Generation 6: Best x = 1.05108 , f(x) = 2.000001
Generation 7: Best x = 1.05108 , f(x) = 2.000001
Generation 8: Best x = 1.05108 , f(x) = 2.000001
Generation 9: Best x = 1.05108 , f(x) = 2.000001
Generation 10: Best x = 1.04998 , f(x) = 2.000000

Final best:
Best x = 1.049980 , f(x) = 2.000000
```

Program 2

Optimization via Gene Expression Algorithm

Algorithm:

Date 4-9-25
Page 1

Lab-3, Optimization in a Genetic Expression Algorithm

- Algo

1. Initialize the population of chromosomes
2. Express the chromosomes into expression trees
3. Evaluate fitness of ET using a fitness function
4. Select chromosome: for reproduction function
5. Select chromosomes for reproduction based on fitness
6. Apply genetic operators :
   1. Mutation
   2. Insertion Sequence Transposition
   3. Root Transposition
   4. One point recombination
   5. Two point recombination
7. Create new population and repeat from Step 2
8. Termination: Stop after a fixed no of generations or if a desired fitness is achieved.

- Pseudocode

```
~~OptimizedGenExpAlgo()~~
~~    Initialize P with N random chromosomes~~
~~    gen = 0~~
~~    Express chromosomes into exp trees T~~
~~    Evaluate fitness using fitFunc()~~
~~    Select~~
OptimizedGenExpAlgo()
    Initialize population with random expr (chromosomes)
    Evaluate fitness of each expression
    while termination condition not met:
            select parents from population (eg.,
    tournament selection, roulette wheel)
            apply crossover operator from
```

PSO

Date
Page 8

parents to offspring

Apply ~~crossover~~ mutation operator to offspring

(small random changes in expⁿ)

Evaluate fitness of offspring

Select next generation population

(elitism + replacement strategy)

if optimization step required:

Apply local optimization

Re-evaluate optimized expⁿ

Update best solution if new best fitness is found

End while

Return best expⁿ found

MG

Code:
#Application:Sequence of Induction & Creation of Constants

```
import random, math

# ===== Sequence Induction Setup =====
# Example sequence: f(n) tries to model this pattern
sequence = [2, 3, 6, 11, 18]   # You can change this
X = list(range(1, len(sequence) + 1))
Y = sequence

# ===== Genome Representation: list tokens like ['x', 3.14, 'x'] =====
TOKENS = ["x"]  # variable name only

def random_genome():
    return ["x"]  # start simple like your notebook

def evaluate(genome, n):
    """Evaluate expression left-to-right: start with n and apply + constant repeatedly."""
    val = n
    try:
        for t in genome[1:]:
            if isinstance(t, (int, float)):
                val = val + t  # only addition for simplicity
        return val
    except:
        return None

def fitness(genome):
    err = 0
    for n, y in zip(X, Y):
        v = evaluate(genome, n)
        if v is None or math.isinf(v):
            return 1e-9
        err += (y - v) ** 2
    return 1 / (1 + err/len(X))

def mutate(genome):
    g = genome[:]
    # Add a new constant
    g.append(round(random.uniform(-6, 6), 6))
    return g

# ===== GEA Execution with Printing Style Matching Your Note =====
def run_gea(iterations=6, test_x=2):
```

```
    genome = random_genome()
    print("Initial Expression:", genome)

    for i in range(iterations):
        genome = mutate(genome)
        print("\nMutated expression:", genome)

        val = evaluate(genome, test_x)
        if val is None:
            print(f"Evaluated at x = {test_x}: None")
        else:
            print(f"Evaluated at x = {test_x}: {round(val,6)}")

    return genome

# ===== Run =====
best = run_gea()
```

```
Initial Expression: ['x']

Mutated expression: ['x', 1.342526]
Evaluated at x = 2: 3.342526

Mutated expression: ['x', 1.342526, 2.327768]
Evaluated at x = 2: 5.670294

Mutated expression: ['x', 1.342526, 2.327768, 5.983546]
Evaluated at x = 2: 11.65384

Mutated expression: ['x', 1.342526, 2.327768, 5.983546, -3.59786]
Evaluated at x = 2: 8.05598

Mutated expression: ['x', 1.342526, 2.327768, 5.983546, -3.59786, -4.371259]
Evaluated at x = 2: 3.684721

Mutated expression: ['x', 1.342526, 2.327768, 5.983546, -3.59786, -4.371259, -5.111376]
Evaluated at x = 2: -1.426655
```

Program 3

Particle Swarm Optimization

Algorithm:

Particle Swarm Optimization

Date 11-9-25
Page 9

Lab-4, ~~Parallel Cellular Optimization~~

→ Algorithm

- Initialize a swarm of particles with random positions and ~~values~~ velocities in the search space
- Each particle evaluates its fitness (objective function value)
- Update:-

cognition — Personal best (pBest) - Best position ever visited by a particle

social influence — Global best (gBest) - Best position ever found by a particle in the swarm

- Update each particle's position & velocity.
- Repeat until a stopping criterion is met (max iterations or acceptable fitness)

→ Pseudo Code

```
Initialize swarm positions x[i] and
velocities v[i]
pBest [i] = x[i] for all particles
gBest = best of pBest
for iter = 1 to max iterations do
    for each particle i do
        v[i] = w * v[i] + c1 * rand() *
(pBest[i] - x[i]) + c2 * rand() * (gBest - x[i])
        x[i] = x[i] + v[i]
        if f(x[i]) < f(pBest[i])
            pBest[i] = x[i]
        end if
    end for
    update gBest = best of all pBest
end for
return gBest
```

MS 11/9/25

Code:

```
#Application: Multi-robot coorporation
import numpy as np

class Particle:
    def __init__(self, dim, bounds):
        self.position = np.random.uniform(bounds[:,0], bounds[:,1], dim)
        self.velocity = np.zeros(dim)
        self.best_pos = np.copy(self.position)
        self.best_cost = float('inf')

def fitness(path, start, goal, obstacles):
    cost = np.linalg.norm(path[-1] - goal)
    # Add obstacle penalty
    for obs in obstacles:
        distances = np.linalg.norm(path - obs['center'], axis=1)
        penalty = np.sum(np.where(distances < obs['radius'], 1000, 0))
        cost += penalty
    return cost

def pso_path_planning(start, goal, obstacles, n_particles=30, dim=10, bounds=None, epochs=100):
    if bounds is None:
        bounds = np.array([[-1, 1]] * dim)  # example bounds
    swarm = [Particle(dim, bounds) for _ in range(n_particles)]
    global_best_pos = None
    global_best_cost = float('inf')

    w = 0.5     # inertia weight
    c1 = c2 = 1.5  # cognitive and social coefficients

    for epoch in range(epochs):
        for p in swarm:
            # Decode particle position into a path (e.g., waypoints) here
            path = [start + (goal - start) * (i / (dim + 1)) + p.position[i] for i in range(dim)]
            path = np.vstack([start, *path, goal])

            cost = fitness(path, start, goal, obstacles)

            if cost < p.best_cost:
                p.best_cost = cost
                p.best_pos = p.position

            if cost < global_best_cost:
                global_best_cost = cost
                global_best_pos = p.position

        for p in swarm:
            r1, r2 = np.random.rand(dim), np.random.rand(dim)
```

```
            p.velocity = w*p.velocity \
                         + c1*r1*(p.best_pos - p.position) \
                         + c2*r2*(global_best_pos - p.position)
            p.position += p.velocity
            # Optionally clip to bounds
            p.position = np.clip(p.position, bounds[:,0], bounds[:,1])

    # Build final path
    best_path = [start + (goal - start) * (i / (dim + 1)) + global_best_pos[i] for i in range(dim)]
    best_path = np.vstack([start, *best_path, goal])
    return best_path, global_best_cost

# Example usage
if __name__ == "__main__":
    start = np.array([0.0, 0.0])
    goal = np.array([10.0, 10.0])
    obstacles = [{'center': np.array([5.0, 5.0]), 'radius': 2.0}]
    path, cost = pso_path_planning(start, goal, obstacles)
    print("Path cost:", cost)
    print("Path points:\n", path)
```

```
Path cost: 0.0
Path points:
 [[ 0.          0.        ]
 [ 1.45720002  1.45720002]
 [ 0.82934424  0.82934424]
 [ 0.30486311  0.30486311]
 [ 1.50984063  1.50984063]
 [ 3.01164867  3.01164867]
 [ 3.48963199  3.48963199]
 [ 6.75135608  6.75135608]
 [ 6.83215368  6.83215368]
 [ 8.02401295  8.02401295]
 [ 6.91166801  6.91166801]
 [10.         10.        ]]
```

Program 4
Ant Colony Optimization

Algorithm:

Lab 5, General Algorithm for Ant Colony

Date 28-10-25
Page (11)

Initialize pheromone values T on all solution components set parameters
α (pheromone influence),
β (heuristic influence),
evaporation rate rho,
no. of ants m

while stopping criteria not met:
for each ant k in 1 to m:
initialize an empty solution sk
while solution s_k is incomplete:
select next solution component c
base on probability proportional to
[T (c)]^α + [n (c)]^β
where n(c) is heuristic desirability of component c
add component c to solution s_k
evaluate the quality of solution s_k

~~for each solution component 'c':~~
~~evaporate phenomena:~~
~~$T(c) = (1-\rho) * T(c)$~~
for each ant k:
deposit pheromone components in solution
$T(c) = T(c) + \Delta T\_k(c)$
amount $\Delta T - k(c)$ depend on quality of solution s_k
return best solution

Code:

```
#Application: Travelling Salesman Problem
import math
import random

# ---------------------------
# Problem definition (TSP)
# ---------------------------

# Example coordinates for 6 cities (you can change these)
coords = [
    (0, 0),
    (2, 6),
    (5, 3),
    (6, 7),
    (8, 3),
    (3, 9)
]

N_CITIES = len(coords)

def euclidean(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

# Distance matrix
dist = [[0.0]*N_CITIES for _ in range(N_CITIES)]
for i in range(N_CITIES):
    for j in range(N_CITIES):
        if i != j:
            dist[i][j] = euclidean(coords[i], coords[j])
        else:
            dist[i][j] = float("inf")

# ---------------------------
# Ant Colony parameters
# ---------------------------

N_ANTS       = N_CITIES         # one ant per city
N_ITER       = 10               # iterations (matches your table length)
ALPHA        = 1.0              # pheromone importance
BETA         = 5.0              # heuristic importance
EVAPORATION  = 0.5              # pheromone evaporation rate
Q            = 100.0            # pheromone deposit factor

random.seed(0)                  # make runs deterministic
```

```
# ---------------------------
# ACO core
# ---------------------------

def construct_solution(pheromone):
    """Build a tour for one ant."""
    n = N_CITIES
    start = random.randrange(n)
    tour = [start]
    unvisited = set(range(n))
    unvisited.remove(start)

    while unvisited:
        i = tour[-1]
        # compute probabilities for each candidate j
        probs = []
        denom = 0.0
        for j in unvisited:
            tau = pheromone[i][j] ** ALPHA
            eta = (1.0 / dist[i][j]) ** BETA
            val = tau * eta
            probs.append((j, val))
            denom += val

        # roulette wheel selection
        r = random.random()
        cum = 0.0
        chosen = None
        for j, val in probs:
            cum += val / denom
            if r <= cum:
                chosen = j
                break
        if chosen is None:
            chosen = list(unvisited)[-1]

        tour.append(chosen)
        unvisited.remove(chosen)

    return tour

def tour_length(tour):
    length = 0.0
    for i in range(len(tour)):
        j = (i + 1) % len(tour)  # return to start
        length += dist[tour[i]][tour[j]]
    return length
```

```
def ant_colony_tsp():
    # initial pheromone
    pheromone = [[1.0 for _ in range(N_CITIES)] for _ in range(N_CITIES)]

    best_tour = None
    best_cost = float("inf")
    best_costs_per_iter = []

    for it in range(1, N_ITER + 1):
        all_tours = []
        all_costs = []

        # build solutions
        for _ in range(N_ANTS):
            tour = construct_solution(pheromone)
            cost = tour_length(tour)
            all_tours.append(tour)
            all_costs.append(cost)

        # update global best
        for tour, cost in zip(all_tours, all_costs):
            if cost < best_cost:
                best_cost = cost
                best_tour = tour

        best_costs_per_iter.append(best_cost)

        # evaporate pheromone
        for i in range(N_CITIES):
            for j in range(N_CITIES):
                pheromone[i][j] *= (1.0 - EVAPORATION)

        # deposit pheromone (Ant System)
        for tour, cost in zip(all_tours, all_costs):
            deposit = Q / cost
            for k in range(len(tour)):
                i = tour[k]
                j = tour[(k + 1) % len(tour)]
                pheromone[i][j] += deposit
                pheromone[j][i] += deposit

    return best_tour, best_costs_per_iter

# ---------------------------
# Run ACO and print like notebook
# ---------------------------

best_tour, best_costs = ant_colony_tsp()
```

```
print("Output")
print("Iteration | Best cost so far =")

for i, c in enumerate(best_costs, start=1):
    print(f"{i:2d}        {c:.6f}")

print(f"\nMin cost = {min(best_costs):.6f}")
print("Best tour (city indices) =", best_tour)
```

```
Output
Iteration | Best cost so far =
 1          26.395472
 2          26.395472
 3          26.395472
 4          26.395472
 5          26.395472
 6          26.395472
 7          26.395472
 8          26.395472
 9          26.395472
10          26.395472

Min cost = 26.395472
Best tour (city indices) = [1, 0, 2, 4, 3, 5]
```

Program 5

Cuckoo Search

Algorithm:

Date 16-10-25
Page 13

Lab-6 Cuckoo Search

Input:

f(x) # Objective func. to min or max
n # no. of nests (population size)
pa # discover probability ($0 \le pa \le 1$)
tmax # max iterations

Optional params: alpha (step-size-scale)
levy-beta (levy exponent)

Output:

x_best, f_best # best sol. found & its fitness

Procedure:

Initialize population

```
nest = InitializeNests(n, bounds)
fitness = EvaluateAll(nest, f)
xbest, fbest = bestOf(nests, fitness)
t = 0
while t < tmax:
    for i = 1 to n:
        x_new = levyflight(nest[i],
                alpha, levy_beta)
        f_new = f(x_new)
```

choose a random nest j & replace it if new solution is better

```
j = RandomInteger(1, n)
if f_new is better than fitness[j]:
    nests[j] = x_new
    fitness[j] = f_new
```

update global best if needed:

x_best = x_new

f_best = f_new

abandon a fraction of worst nests & generate a new random ones

numabandon = ceil (pa * n)

return f_best, x_best

Code:

```
#Application:Optimization of Traffic Management
import random
import math

# ---------------- Traffic Simulation Function ---------------- #

def simulate_traffic(green_times):
    """
    green_times: [g1, g2, g3, g4] green time for each of 4 directions
    returns total waiting time (to be minimized)
    """

    # average traffic arrival rate (vehicles/sec) for each direction
    arrival = [0.8, 1.1, 0.6, 0.9]  # adjust as required

    total_waiting = 0

    for arr, green in zip(arrival, green_times):
        # vehicles arriving during a cycle
        vehicles = arr * cycle_time

        # vehicles that can pass (assume 1 vehicle/second during green)
        passed = green

        # queue leftover -> waiting contribution
        waiting = max(0, vehicles - passed)

        # weighted wait time approximation
        total_waiting += waiting**2

    return total_waiting

# ---------------- Cuckoo Search Optimization ---------------- #

def levy_flight(Lambda=1.5):
    u = random.random() * 0.1
    v = random.random()
    step = u / (abs(v) ** (1 / Lambda))
    return step

def cuckoo_search(n=10, pa=0.25, iterations=50):
    # initial nests (green times for 4 signals)
    nests = [[random.uniform(10, 60) for _ in range(4)] for _ in range(n)]

    fitness = [simulate_traffic(nest) for nest in nests]

    global_best = nests[fitness.index(min(fitness))]
    global_best_fit = min(fitness)
```

```
    for it in range(iterations):

        # Generate new solutions by Levy flights
        for i in range(n):
            new = []
            for x in nests[i]:
                step = levy_flight()
                new_val = x + step * (x - global_best[0])
                new.append(max(5, min(90, new_val)))  # bounds
            new_fit = simulate_traffic(new)

            if new_fit < fitness[i]:
                nests[i], fitness[i] = new, new_fit

                if new_fit < global_best_fit:
                    global_best = new
                    global_best_fit = new_fit

        # Abandon some worst nests
        for i in range(n):
            if random.random() < pa:
                nests[i] = [random.uniform(10, 60) for _ in range(4)]
                fitness[i] = simulate_traffic(nests[i])

        print(f"Iteration {it+1} | Best waiting time = {global_best_fit:.4f}")

    return global_best, global_best_fit

# ---------------- Execution ---------------- #

cycle_time = 120  # total cycle length in seconds (modifiable)

best_solution, best_wait = cuckoo_search()

print("\nOptimal Green Signal Timings (seconds):")
print(f"North-South Straight   : {best_solution[0]:.2f}")
print(f"North-South Right Turn : {best_solution[1]:.2f}")
print(f"East-West Straight     : {best_solution[2]:.2f}")
print(f"East-West Right Turn   : {best_solution[3]:.2f}")

print(f"\nMinimum Estimated Waiting Time: {best_wait:.4f}")
```

```
Iteration 1 | Best waiting time = 12718.1259
Iteration 2 | Best waiting time = 12718.1259
Iteration 3 | Best waiting time = 12718.1259
Iteration 4 | Best waiting time = 12718.1259
Iteration 5 | Best waiting time = 12718.1259
Iteration 6 | Best waiting time = 12718.1259
Iteration 7 | Best waiting time = 12718.1259
Iteration 8 | Best waiting time = 12718.1259
Iteration 9 | Best waiting time = 12718.1259
Iteration 10 | Best waiting time = 12718.1259
Iteration 11 | Best waiting time = 12718.1259
Iteration 12 | Best waiting time = 12718.1259
Iteration 13 | Best waiting time = 12718.1259
Iteration 14 | Best waiting time = 12718.1259
Iteration 15 | Best waiting time = 12718.1259
Iteration 16 | Best waiting time = 12718.1259
Iteration 17 | Best waiting time = 12718.1259
Iteration 18 | Best waiting time = 12718.1259
Iteration 19 | Best waiting time = 12718.1259
Iteration 20 | Best waiting time = 12718.1259
Iteration 21 | Best waiting time = 12718.1259
Iteration 22 | Best waiting time = 12718.1259
Iteration 23 | Best waiting time = 12718.1259
Iteration 24 | Best waiting time = 12718.1259
Iteration 25 | Best waiting time = 12718.1259
Iteration 26 | Best waiting time = 12718.1259
Iteration 27 | Best waiting time = 12718.1259
Iteration 28 | Best waiting time = 12718.1259
Iteration 29 | Best waiting time = 12718.1259
Iteration 30 | Best waiting time = 12718.1259
```

Program 6

Grey Wolf Optimization

Algorithm:

classmate
Date 23-10-25
Page 15

Grey Wolf Optimizer

Algorithm

1. Define the function f(x)
2. Initialize parameters:
   - N = No. of wolves
   - maxiteration = max. no. of iterations
3. Initialize the position of N wolves randomly within search space bounds
4. Evaluate the fitness of each wolf using f(x)
5. We ~~second~~ select 3 best wolves:
   - α: Best wolf
   - β: 2nd best wolf
   - γ: 3rd best wolf
6. for t = 1 to maxIterations:

    for each wolf:

        generate random no. r1, r2 ∈ [0, 1]

        Compute:

            $A = 2 * A * r1 - A$

            $C = 2 * r2$

        for each dimension j:

            $D\alpha = |C * \alpha_j - x_{ij}|$

            $D\beta = |C * \beta_j - x_{ij}|$

            $D\delta = |C * \delta_j - x_{ij}|$

            $x1 = \alpha_j - A * D\alpha$

            $x2 = \beta_j - A * D\beta$

            $x3 = \delta_j - A - D\delta$

            $x_{ij} = (x1 + x2 + x3)/3$

        end for

    end for

IP

Black & white contrast Enhancement

update $\alpha = \alpha - \xi\, 2\, \alpha t$ / maxIteration)

Recalculate fitness

update α, β, δ

7. Return α as the best soln

End

Best parameters $\rightarrow$ $\alpha = 1.25$, $\beta = -0.08$, $\delta = 1.65$

objective (entropy) : -6.8123

MG.

Code:

```
#Application: Black and White Contrast Exhaust
import os
import cv2
import numpy as np
import random
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

# ---------- Choose filename ----------

# ---------- Load image ----------
img = cv2.imread(filename, 0)   # 0 -> grayscale
print("Trying to load:", filename)
if img is None:
    print("❌ Image NOT loaded. Files in current directory:")
    print(os.listdir())
    raise SystemExit("Make sure 'bis.jpg' is uploaded (Files panel) or rename the filename variable.")
else:
    print("✅ Image loaded. shape:", img.shape)

# ---------- Display original (quick) ----------
print("\nOriginal image (quick preview):")
cv2_imshow(img)

# ---------- Functions used by GWO ----------
def image_fitness(img):
    """Fitness = entropy + std (higher is better)."""
    hist = cv2.calcHist([img],[0],None,[256],[0,256])
    hist_norm = hist.ravel() / (hist.sum() + 1e-12)
    entropy = -np.sum(hist_norm * np.log2(hist_norm + 1e-12))
    std = np.std(img)
    return float(entropy + std)

def apply_gamma(img, gamma):
    """Apply gamma correction safely and return uint8 image."""
    corrected = np.power(img.astype(np.float32) / 255.0, gamma)
    corrected = np.uint8(np.clip(corrected * 255.0, 0, 255))
    return corrected

# ---------- Grey Wolf Optimizer (robust init) ----------
def gwo_contrast(img, wolves=8, iterations=20):
    lb, ub = 0.1, 3.0
    population = np.random.uniform(lb, ub, wolves)

    def fit(g):
        return image_fitness(apply_gamma(img, g))

    # initial fitnesses
```

```
    fitness = np.array([fit(g) for g in population])
    # initialize alpha/beta/delta from top-3 fitness values
    idx = np.argsort(-fitness)
    alpha, alpha_f = float(population[idx[0]]), float(fitness[idx[0]])
    beta,  beta_f  = float(population[idx[1]]) if wolves>1 else alpha, float(fitness[idx[1]]) if wolves>1 else alpha_f
    delta, delta_f = float(population[idx[2]]) if wolves>2 else alpha, float(fitness[idx[2]]) if wolves>2 else alpha_f

    for t in range(iterations):
        a = 2 * (1 - t / iterations)
        for i in range(wolves):
            r1, r2 = random.random(), random.random()
            A1 = 2*a*r1 - a; C1 = 2*r2
            r1, r2 = random.random(), random.random()
            A2 = 2*a*r1 - a; C2 = 2*r2
            r1, r2 = random.random(), random.random()
            A3 = 2*a*r1 - a; C3 = 2*r2

            # compute candidate positions (safe: alpha/beta/delta are floats)
            X1 = alpha - A1 * abs(C1*alpha - population[i])
            X2 = beta  - A2 * abs(C2*beta  - population[i])
            X3 = delta - A3 * abs(C3*delta - population[i])

            population[i] = np.clip((X1 + X2 + X3) / 3.0, lb, ub)
            new_fit = fit(population[i])

            # update alpha/beta/delta
            if new_fit > alpha_f:
                delta, delta_f = beta, beta_f
                beta, beta_f = alpha, alpha_f
                alpha, alpha_f = float(population[i]), float(new_fit)
            elif new_fit > beta_f:
                delta, delta_f = beta, beta_f
                beta, beta_f = float(population[i]), float(new_fit)
            elif new_fit > delta_f:
                delta, delta_f = float(population[i]), float(new_fit)

        print(f"Iteration {t+1}/{iterations} — Best Gamma = {alpha:.4f} | Fitness = {alpha_f:.4f}")

    # return enhanced image and numeric gamma
    return apply_gamma(img, float(alpha)), float(alpha)

# ---------- Run optimizer ----------
enhanced, best_gamma = gwo_contrast(img, wolves=8, iterations=20)
print("\n🎯 Optimal Gamma:", best_gamma)

# ---------- Display original and enhanced side-by-side ----------
orig_display = img.copy()
```

```
enh_display = enhanced.copy()

# Use matplotlib to show both images in one row
plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.title("Original (bis.jpg)")
plt.axis('off')
plt.imshow(orig_display, cmap='gray')

plt.subplot(1,2,2)
plt.title(f"Enhanced (gamma={best_gamma:.3f})")
plt.axis('off')
plt.imshow(enh_display, cmap='gray')

plt.show()
```

```
Trying to load: bis.jpg
✅ Image loaded. shape: (183, 275)

Original image (quick preview):
```

```
Iteration 1/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 2/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 3/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 4/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 5/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 6/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 7/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 8/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 9/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 10/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 11/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 12/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 13/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 14/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 15/20 – Best Gamma = 3.0000 | Fitness = 81.8786
Iteration 16/20 – Best Gamma = 2.9905 | Fitness = 81.8812
Iteration 17/20 – Best Gamma = 2.9905 | Fitness = 81.8812
```

```
Iteration 18/20 — Best Gamma = 2.9992 | Fitness = 81.8896
Iteration 19/20 — Best Gamma = 2.9992 | Fitness = 81.8896
Iteration 20/20 — Best Gamma = 2.9992 | Fitness = 81.8896

🎯 Optimal Gamma: 2.999225234597938
```

Program 7
Parallel Cellular Algorithm

Algorithm:

Parallel Cellular Algo.

Date 30/10/25
Page 17

1. Define the optimization problem function f(x) to be minimized or maximized

2. numcells = total no. of cells
grid structure < 1D or 2D>
neighbourhood type
maxiteration — max. no of iteration
α — constant

3. Initialize population. For each cell i in grid do
position [i] ← random pos in search space
fitness [i] ← f(position [i])

End for

4. Main loop
for t = 1 to maxiterations do in parallel
for each cell i do neighbours ← getneigh (i, type)

bestneighbour ← neigh with the best fitness among neigh

5. Update state of cells
position [i] ← position [i] + α f(bestneigh. position − position [i])

6. Evaluate new fitness

fitness [i] ← f (position [i])

End for

Synchronize updated position & fitness value across the grid

End for

best cell ← cell with best fitness

return best cell position & fitness

End

Application → Routing & Scheduling

Code:

```
#Application: Routing and Scheduling
import random
import math
from copy import deepcopy

# ================================
# Problem Definition
# ================================

# Depot + customers (x, y)
# index 0 = depot
coords = [
    (0, 0),   # depot
    (2, 3),   # customer 1
    (5, 4),   # customer 2
    (6, 1),   # customer 3
    (8, 3),   # customer 4
    (1, 6)    # customer 5
]

n_customers = len(coords) - 1
customers = list(range(1, n_customers + 1))

# service times for each node (0 unused)
service_time = [0, 3, 4, 2, 3, 5]

# simple time windows (ready_time, due_time) for each node
# depot not used
ready_time = [0, 0,  0,  0,  0,  0]
due_time   = [0, 30, 40, 35, 45, 50]

# travel time = Euclidean distance (you can scale if needed)
def distance(i, j):
    (x1, y1) = coords[i]
    (x2, y2) = coords[j]
    return math.hypot(x2 - x1, y2 - y1)


# ================================
# Evaluation: Routing + Scheduling
# ================================

def evaluate_route(route, big_penalty=1000):
    """
    route: list of customers (permutation)
    Returns: (cost, schedule)
      - cost: total distance + penalties for time window violations
      - schedule: list of dicts with arrival/start/finish times
```

```
    """
    time = 0.0
    pos = 0  # start at depot
    total_dist = 0.0
    total_lateness = 0.0
    schedule = []

    for c in route:
        # travel to customer
        travel = distance(pos, c)
        time += travel
        total_dist += travel

        arrival = time

        # wait if early
        if time < ready_time[c]:
            wait = ready_time[c] - time
            time += wait

        start_service = time
        time += service_time[c]
        finish = time

        # lateness if finish after due time
        lateness = max(0.0, finish - due_time[c])
        total_lateness += lateness

        schedule.append({
            "customer": c,
            "arrival": round(arrival, 2),
            "start": round(start_service, 2),
            "finish": round(finish, 2),
            "lateness": round(lateness, 2)
        })

        pos = c

    # return to depot
    travel_back = distance(pos, 0)
    total_dist += travel_back

    cost = total_dist + big_penalty * total_lateness
    return cost, schedule


# ===============================
# Parallel Cellular Algorithm
# ===============================
```

```python
# 2D grid of cells (each holds one route)
GRID_ROWS = 4
GRID_COLS = 4
N_ITER = 50
MUTATION_RATE = 0.3


def random_route():
    r = customers[:]
    random.shuffle(r)
    return r


def neighbors_indices(r, c, rows, cols):
    """Moore neighborhood with wrap-around (toroidal)."""
    neigh = []
    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            if dr == 0 and dc == 0:
                continue
            nr = (r + dr) % rows
            nc = (c + dc) % cols
            neigh.append((nr, nc))
    return neigh


def local_mutation(route):
    """Simple mutation: swap two customers."""
    new_route = route[:]
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(len(new_route)), 2)
        new_route[i], new_route[j] = new_route[j], new_route[i]
    return new_route


def parallel_cellular_routing():
    # initialize grid with random routes
    grid = [[random_route() for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
    grid_costs = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]

    # evaluate initial population
    global_best_route = None
    global_best_cost = float("inf")

    for i in range(GRID_ROWS):
        for j in range(GRID_COLS):
            cost, _ = evaluate_route(grid[i][j])
            grid_costs[i][j] = cost
```

```
        if cost < global_best_cost:
            global_best_cost = cost
            global_best_route = deepcopy(grid[i][j])

print("Initial best cost:", round(global_best_cost, 2), "route:", global_best_route)

# main iterations
for it in range(1, N_ITER + 1):
    new_grid = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]
    new_costs = [[None for _ in range(GRID_COLS)] for _ in range(GRID_ROWS)]

    # update all cells "in parallel"
    for r in range(GRID_ROWS):
        for c in range(GRID_COLS):
            # select best neighbor (including itself)
            best_local_route = grid[r][c]
            best_local_cost = grid_costs[r][c]

            for nr, nc in neighbors_indices(r, c, GRID_ROWS, GRID_COLS):
                if grid_costs[nr][nc] < best_local_cost:
                    best_local_cost = grid_costs[nr][nc]
                    best_local_route = grid[nr][nc]

            # produce new route by mutating the best neighbor

            child_route = local_mutation(best_local_route)
            child_cost, _ = evaluate_route(child_route)

            # choose better between parent and child for this cell
            if child_cost < best_local_cost:
                new_grid[r][c] = child_route
                new_costs[r][c] = child_cost
            else:
                new_grid[r][c] = best_local_route[:]
                new_costs[r][c] = best_local_cost

            # update global best
            if new_costs[r][c] < global_best_cost:
                global_best_cost = new_costs[r][c]
                global_best_route = deepcopy(new_grid[r][c])

    grid, grid_costs = new_grid, new_costs

    if it % 5 == 0 or it == 1:
        print(f"Iteration {it:2d} | Global best cost = {round(global_best_cost, 2)}")

# final evaluation for schedule
best_cost, best_schedule = evaluate_route(global_best_route)
return global_best_route, best_cost, best_schedule
```

```
# ================================
# Run Example
# ================================

if __name__ == "__main__":
    random.seed(0)

    best_route, best_cost, schedule = parallel_cellular_routing()

    print("\n=== FINAL BEST SOLUTION ===")
    print("Best route (customer order):", best_route)
    print("Best cost:", round(best_cost, 2))
    print("\nSchedule (Routing + Timing):")
    print("Cust | Arrival | Start | Finish | Lateness")
    for s in schedule:
        print(f"{s['customer']:4d} | {s['arrival']:7.2f} | {s['start']:5.2f} | "
              f"{s['finish']:6.2f} | {s['lateness']:8.2f}")
```

```
Initial best cost: 28.1 route: [1, 3, 2, 4, 5]
Iteration  1 | Global best cost = 23.31
Iteration  5 | Global best cost = 23.31
Iteration 10 | Global best cost = 23.31
Iteration 15 | Global best cost = 23.31
Iteration 20 | Global best cost = 23.31
Iteration 25 | Global best cost = 23.31
Iteration 30 | Global best cost = 23.31
Iteration 35 | Global best cost = 23.31
Iteration 40 | Global best cost = 23.31
Iteration 45 | Global best cost = 23.31
Iteration 50 | Global best cost = 23.31

=== FINAL BEST SOLUTION ===
Best route (customer order): [1, 5, 2, 4, 3]
Best cost: 23.31

Schedule (Routing + Timing):
Cust | Arrival | Start | Finish | Lateness
   1 |    3.61 |  3.61 |   6.61 |     0.00
   5 |    9.77 |  9.77 |  14.77 |     0.00
   2 |   19.24 | 19.24 |  23.24 |     0.00
   4 |   26.40 | 26.40 |  29.40 |     0.00
   3 |   32.23 | 32.23 |  34.23 |     0.00
```