VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Krithika H Kotian (1BM23CS159)

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 **Krithika H Kotian(1BM23CS159),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

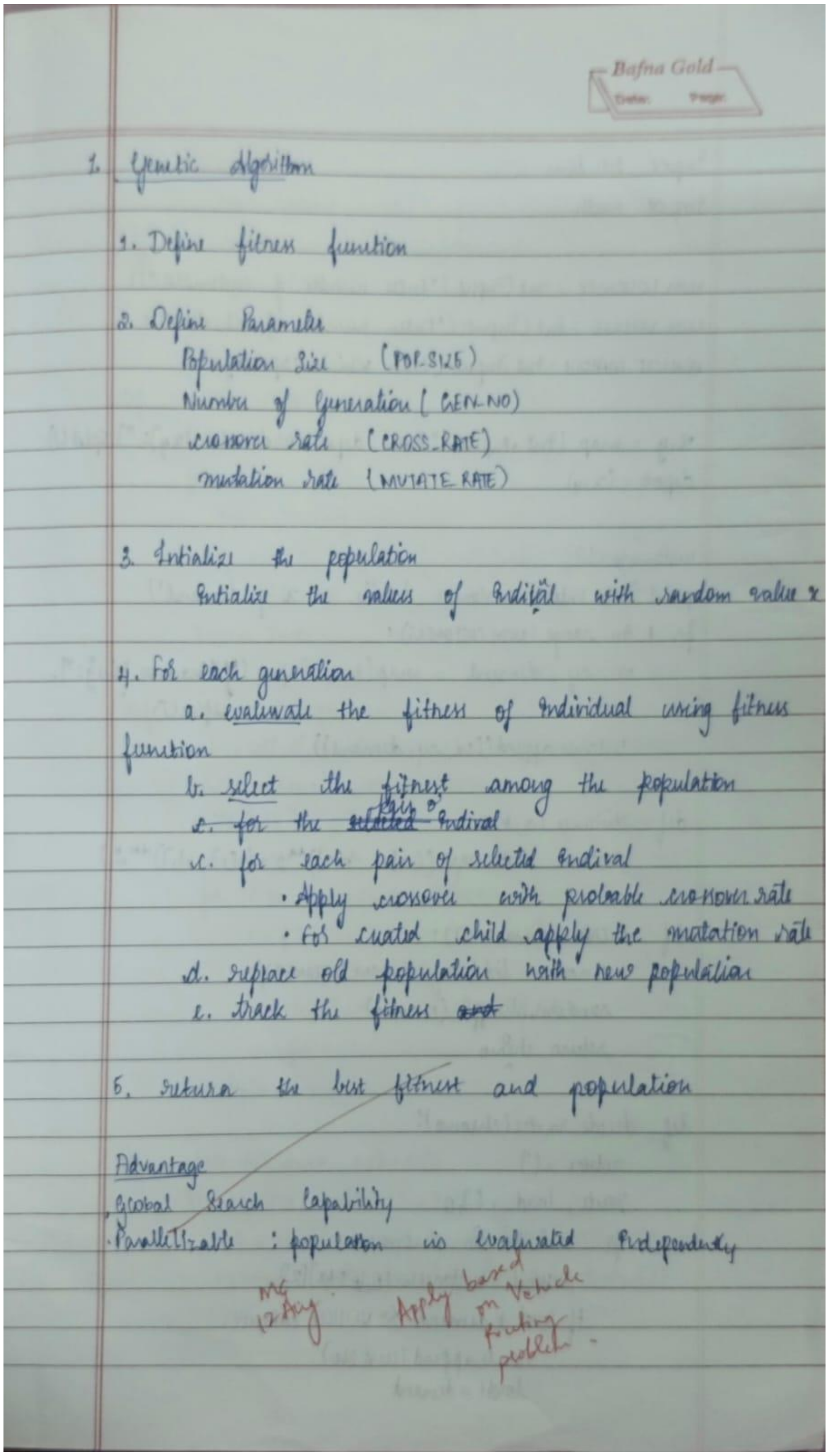
Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/krithikahkotian/Bio-Inspired-Systems

Program 1

Genetic Algorithm for Optimization Problems



1. Genetic Algorithm

1. Define fitness function

2. Define Parameter
Population Size (POP.SIZE)
Number of Generation (GEN-NO)
crossover rate (CROSS_RATE)
mutation rate (MUTATE_RATE)

3. Initialize the population
Initialize the values of Individual with random value x

4. For each generation
a. evaluate the fitness of Individual using fitness function
b. select the fittest among the population
~~c. for the selected~~ pair indival
c. for each pair of selected indival
• Apply crossover with probable crossover rate
• for created child apply the mutation rate
d. replace old population with new population
e. track the fitness ~~and~~

5. return the best fitness and population

Advantage
Global Search capability
Parallelizable : population is evaluated independently

MG 12 Aug.
Apply based on Vehicle Routing problem.

```
import math
import random
```

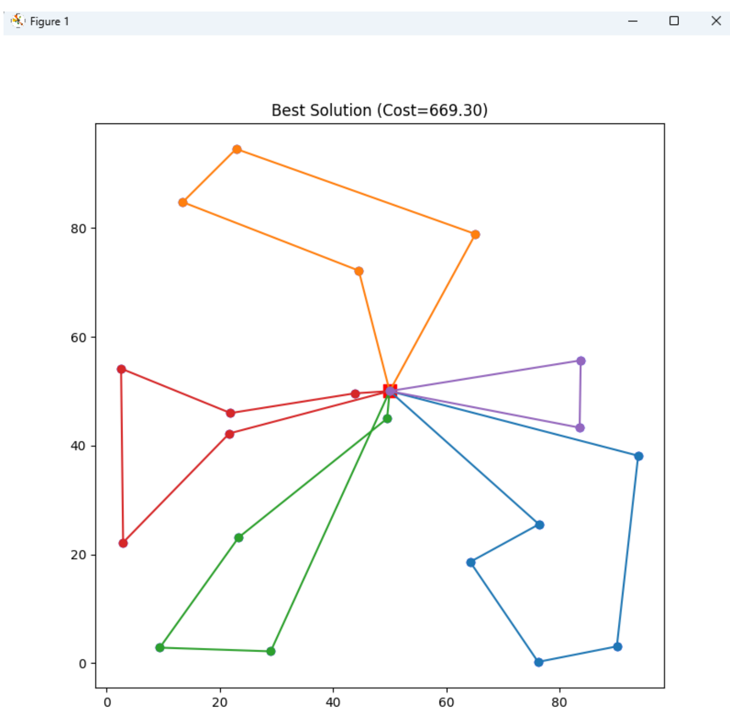
```
import matplotlib.pyplot as plt

# ---------------- Problem Setup ----------------
NUM_CUSTOMERS = 20     # number of customers
VEHICLE_CAPACITY = 30  # max load per vehicle
POP_SIZE = 80          # population size
GENERATIONS = 300
TOURNAMENT_K = 3
CROSSOVER_RATE = 0.9
MUTATION_RATE = 0.2
ELITE_SIZE = 2

random.seed(1)

# Depot
depot = (50, 50)

# Customers (randomly generated)
customers = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(NUM_CUSTOMERS)]
demands = [random.randint(1, 10) for _ in range(NUM_CUSTOMERS)]

# Distance matrix
points = [depot] + customers
n = len(points)
dist = [[0.0]*n for _ in range(n)]
for i in range(n):
    for j in range(n):
        dx, dy = points[i][0] - points[j][0], points[i][1] - points[j][1]
        dist[i][j] = math.hypot(dx, dy)

# ---------------- Decoder ----------------
def decode_permutation(perm):
    routes, cur_route, cur_load = [], [], 0
    for c in perm:
        d = demands[c-1]
        if cur_load + d <= VEHICLE_CAPACITY:
            cur_route.append(c)
            cur_load += d
        else:
            routes.append(cur_route)
            cur_route, cur_load = [c], d
    if cur_route:
        routes.append(cur_route)
    return routes

def route_cost(route):
    if not route:
```

```
        return 0
    cost = dist[0][route[0]]
    for i in range(len(route)-1):
        cost += dist[route[i]][route[i+1]]
    cost += dist[route[-1]][0]
    return cost

def total_cost(routes):
    return sum(route_cost(r) for r in routes)

# ---------------- Genetic Operators ----------------
def random_permutation():
    perm = list(range(1, NUM_CUSTOMERS+1))
    random.shuffle(perm)
    return perm

def tournament_selection(pop, fitness):
    best = None
    for _ in range(TOURNAMENT_K):
        ind = random.choice(pop)
        if best is None or fitness[tuple(ind)] < fitness[tuple(best)]:
            best = ind
    return best[:]

def ordered_crossover(p1, p2):
    n = len(p1)
    a, b = sorted(random.sample(range(n), 2))
    def ox(x, y):
        child = [-1]*n
        child[a:b+1] = x[a:b+1]
        pos = (b+1) % n
        for elem in y:
            if elem not in child:
                child[pos] = elem
                pos = (pos+1) % n
        return child
    return ox(p1, p2), ox(p2, p1)

def swap_mutation(perm):
    a, b = random.sample(range(len(perm)), 2)
    perm[a], perm[b] = perm[b], perm[a]

# ---------------- GA ----------------
population = [random_permutation() for _ in range(POP_SIZE)]
fitness = {}

best_cost = float("inf")
```

```
best_solution = None
history = []

for gen in range(GENERATIONS):
    # evaluate
    for ind in population:
        if tuple(ind) not in fitness:
            routes = decode_permutation(ind)
            fitness[tuple(ind)] = total_cost(routes)
    # track best
    for ind in population:
        cost = fitness[tuple(ind)]
        if cost < best_cost:
            best_cost = cost
            best_solution = ind[:]
    history.append(best_cost)

    # elitism
    sorted_pop = sorted(population, key=lambda x: fitness[tuple(x)])
    new_pop = sorted_pop[:ELITE_SIZE]

    # reproduction
    while len(new_pop) < POP_SIZE:
        p1, p2 = tournament_selection(population, fitness), tournament_selection(population, fitness)
        if random.random() < CROSSOVER_RATE:
            c1, c2 = ordered_crossover(p1, p2)
        else:
            c1, c2 = p1, p2
        if random.random() < MUTATION_RATE: swap_mutation(c1)
        if random.random() < MUTATION_RATE: swap_mutation(c2)
        new_pop.extend([c1, c2])
    population = new_pop[:POP_SIZE]

# ---------------- Results ----------------
best_routes = decode_permutation(best_solution)
print(f"Best cost: {best_cost:.2f}")
for i, r in enumerate(best_routes, 1):
    load = sum(demands[c-1] for c in r)
    print(f"Route {i}: {r}, load={load}, cost={route_cost(r):.2f}")

# Plot solution
plt.figure(figsize=(8,8))
plt.scatter([p[0] for p in customers], [p[1] for p in customers], c='blue')
plt.scatter(*depot, c='red', marker='s', s=100)
for route in best_routes:
    xs = [depot[0]] + [points[c][0] for c in route] + [depot[0]]
    ys = [depot[1]] + [points[c][1] for c in route] + [depot[1]]
```

```
    plt.plot(xs, ys, marker='o')
plt.title(f"Best Solution (Cost={best_cost:.2f})")
plt.show()

# Plot convergence
plt.plot(history)
plt.title("Convergence")
plt.xlabel("Generation")
plt.ylabel("Best Cost")
plt.show()
```

```
[Running] python -u "c:\Users\BMSCE\Documents\1BM23CS159\BIS\LAB1\vehicle_routing_problem.py"
Best cost: 669.30
Route 1: [12, 10, 7, 20, 2], load=28, cost=166.88
Route 2: [8, 1, 9, 4], load=24, cost=147.66
Route 3: [18, 5, 16, 3], load=28, cost=135.67
Route 4: [13, 14, 11, 17, 15], load=30, cost=138.24
Route 5: [19, 6], load=16, cost=80.84
```



Program 2:

Particle Swarm Optimization for Function Optimization

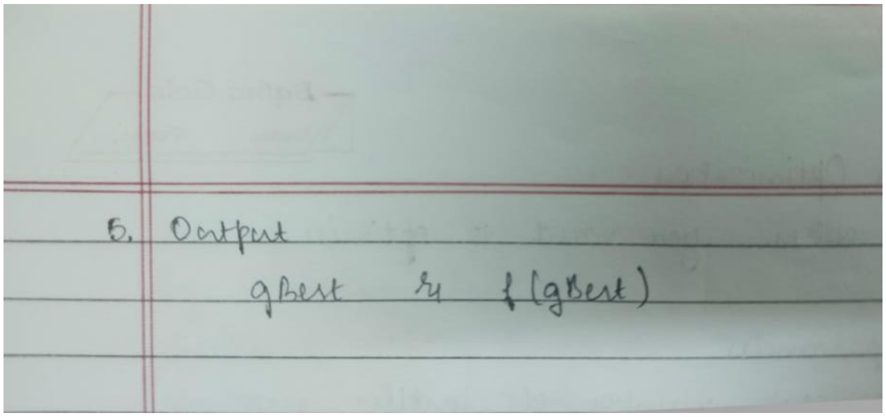
Particle Swarm Optimization

1. Define the problem you want to optimise

2. ~~Initial~~ Define parameter
   num particle : Total number of particle
   position [i] , velocity [i] : position & velocity of each part.
   num_iteration : Total number of iteration
   w : inertia
   c1, c2 : cognitive co-efficient
   pbestposition [i] : The best position of each particle
   gbestposition : The best position of all particle

3. Initialize
   for every particle in num_particle
   initialize position [i] , velocity [i] randomly
   pBestposition[i] = position[i]
   ~~choose the~~ best pBestposition
   f(Bestposition) = value
   Select the best value & initialize its position to gBest position.

4. Iterate
   • for each iteration t=1 to num iteration ;
   for each particle i=1 to num of particle
   a) $v_i = w \cdot v_i + c_1 \cdot r_1 \cdot (p_i - x_i) + c_2 r_2 (g - x_i)$
   b. p[i+1] = p[i] + v[i]
   c. Evaluate fitness
   if f(p[i]) > f(pBest[i])
   pBest[i] = p[i];
   if f(p[i]) > f(gBest)
   gBest[i] = p[i];




```
import cv2
import numpy as np
import os
from skimage.measure import shannon_entropy
from skimage.filters import sobel


# ---------------- Fitness Function ----------------
def fitness_function(image, alpha=0.7, beta=0.3):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    entropy = shannon_entropy(gray)
    edges = sobel(gray)
    edge_strength = np.mean(edges)
    return alpha * entropy + beta * edge_strength


# ---------------- Image Enhancement ----------------
def enhance_image(img, contrast, brightness):
    return cv2.convertScaleAbs(img, alpha=contrast, beta=brightness)


# ---------------- Particle Swarm Optimization ----------------
def pso_optimize(image, num_particles=15, max_iter=30, alpha=0.7, beta=0.3):
    contrast_range = (0.5, 3.0)
    brightness_range = (-50, 50)

    particles = np.random.rand(num_particles, 2)
    particles[:, 0] = contrast_range[0] + particles[:, 0] * (contrast_range[1] - contrast_range[0])
    particles[:, 1] = brightness_range[0] + particles[:, 1] * (brightness_range[1] - brightness_range[0])

    velocities = np.random.uniform(-1, 1, (num_particles, 2))

    personal_best_positions = particles.copy()
    personal_best_scores = np.array([
        fitness_function(enhance_image(image, c, b), alpha, beta)
```

```
        for c, b in particles
    ])

    global_best_index = np.argmax(personal_best_scores)
    global_best_position = personal_best_positions[global_best_index].copy()
    global_best_score = personal_best_scores[global_best_index]

    w, c1, c2 = 0.7, 1.5, 1.5

    for iteration in range(max_iter):
        for i in range(num_particles):
            r1, r2 = np.random.rand(2)
            velocities[i] = (
                w * velocities[i]
                + c1 * r1 * (personal_best_positions[i] - particles[i])
                + c2 * r2 * (global_best_position - particles[i])
            )
            particles[i] += velocities[i]

            particles[i, 0] = np.clip(particles[i, 0], *contrast_range)
            particles[i, 1] = np.clip(particles[i, 1], *brightness_range)

            enhanced = enhance_image(image, particles[i, 0], particles[i, 1])
            score = fitness_function(enhanced, alpha, beta)

            if score > personal_best_scores[i]:
                personal_best_scores[i] = score
                personal_best_positions[i] = particles[i].copy()

        best_idx = np.argmax(personal_best_scores)
        if personal_best_scores[best_idx] > global_best_score:
            global_best_score = personal_best_scores[best_idx]
            global_best_position = personal_best_positions[best_idx].copy()

        print(f"Iteration {iteration+1}/{max_iter}, Best Score: {global_best_score:.4f}")

    return global_best_position, global_best_score


# ---------------- Interactive Run ----------------
def main():
    # Ask user for input image
    input_path = input("Enter input image file name (with extension, e.g., input.jpg): ").strip()

    if not os.path.exists(input_path):
        print(f"Error: File '{input_path}' not found.")
        return
```

```
    # Auto-generate output file name
    base, ext = os.path.splitext(input_path)
    output_path = f"{base}_enhanced{ext}"

    # Load image
    image = cv2.imread(input_path)

    # Run PSO
    best_params, best_score = pso_optimize(image)
    best_contrast, best_brightness = best_params
    print(f"Optimal Contrast: {best_contrast:.3f}, Optimal Brightness: {best_brightness:.3f}")

    # Apply enhancement
    enhanced = enhance_image(image, best_contrast, best_brightness)

    # Save output
    cv2.imwrite(output_path, enhanced)
    print(f"Enhanced image saved as {output_path}")


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

Input Image:

Output Image:

Program 3

Ant Colony Optimization for the Traveling Salsman Problem:

Ant Colony Optimisation:

Input:

- Set of cities
- Number of ants (m)
- The Pheromone rate (α)
- Path Distance
- Number of Iteration
- evaporation rate (ρ)
- heuristic importance (β)

Output: Best Path

Best path length

1. Initialize

Compute distance matrix d[i][j] between all cities

Initialize pheromone matrix $\tau[i][j] = \tau_0$

Best length = 999(∞) & Best_path = None (0)

2. For each iteration (1 . . . N_iteration)

a. For each ant (1 . . . m):

Place ant on a random starting city

Initialize visited = {start} and path = [start]

While not all cities visited:

- From current city i, compute transition probability

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\Sigma (\tau_{ik})^{\alpha} \cdot (\eta_{ik})^{\beta}}$$

- Select next city based on probability
- Add city to path and mark visited

- Close tour by returning to start city
- Compute total tour length 'L'

TSP

```
import random

class ACO_TSP:
    def __init__(self, graph, pheromone, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.5, Q=100):
        """
        graph       : adjacency matrix of distances (2D list)
        pheromone  : initial pheromone matrix (2D list)
        n_ants      : number of ants
        n_iterations : number of iterations
        alpha       : pheromone importance
        beta        : heuristic (1/distance) importance
        rho         : evaporation rate
        Q           : pheromone deposit factor
        """
        self.graph = graph
        self.pheromone = pheromone
        self.n = len(graph)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q

    def run(self):
        best_length = float('inf')
        best_path = None

        for it in range(self.n_iterations):
            all_paths = []
            all_lengths = []

            for ant in range(self.n_ants):
                path = self.construct_solution()
                length = self.path_length(path)
                all_paths.append(path)
                all_lengths.append(length)

                if length < best_length:
                    best_length = length
                    best_path = path

            self.update_pheromones(all_paths, all_lengths)
            print(f"Iteration {it+1}/{self.n_iterations} - Best Length: {best_length:.2f}")

        return best_path, best_length
```

```
def construct_solution(self):
    start = random.randint(0, self.n - 1)
    path = [start]
    visited = set(path)

    while len(path) < self.n:
        current = path[-1]
        next_city = self.choose_next_city(current, visited)
        path.append(next_city)
        visited.add(next_city)

    return path

def choose_next_city(self, current, visited):
    probabilities = []
    denominator = 0

    for j in range(self.n):
        if j not in visited:
            tau = self.pheromone[current][j] ** self.alpha
            eta = (1 / self.graph[current][j]) ** self.beta if self.graph[current][j] > 0 else 0
            denominator += tau * eta

    for j in range(self.n):
        if j not in visited:
            tau = self.pheromone[current][j] ** self.alpha
            eta = (1 / self.graph[current][j]) ** self.beta if self.graph[current][j] > 0 else 0
            probabilities.append((j, (tau * eta) / denominator))

    r = random.random()
    cumulative = 0
    for city, prob in probabilities:
        cumulative += prob
        if r <= cumulative:
            return city

    return probabilities[-1][0]  # fallback

def path_length(self, path):
    length = 0
    for i in range(len(path) - 1):
        length += self.graph[path[i]][path[i + 1]]
    length += self.graph[path[-1]][path[0]]  # return to start
    return length

def update_pheromones(self, all_paths, all_lengths):
    # Evaporation
```

```
        for i in range(self.n):
            for j in range(self.n):
                self.pheromone[i][j] *= (1 - self.rho)

        # Deposit
        for path, length in zip(all_paths, all_lengths):
            deposit = self.Q / length
            for i in range(len(path) - 1):
                a, b = path[i], path[i + 1]
                self.pheromone[a][b] += deposit
                self.pheromone[b][a] += deposit
            # closing edge
            self.pheromone[path[-1]][path[0]] += deposit
            self.pheromone[path[0]][path[-1]] += deposit


# ---------------- Example Usage ----------------
if __name__ == "__main__":
    # Example Graph (distance matrix)
    graph = [
        [0, 10, 12, 11],
        [10, 0, 13, 15],
        [12, 13, 0, 9],
        [11, 15, 9, 0]
    ]

    # Example Initial Pheromone Matrix
    pheromone = [
        [0, 1, 1, 1],
        [1, 0, 1, 1],
        [1, 1, 0, 1],
        [1, 1, 1, 0]
    ]

    aco = ACO_TSP(graph, pheromone, n_ants=5, n_iterations=20, alpha=1, beta=5, rho=0.5, Q=100)
    best_path, best_length = aco.run()

    print("\nBest Path Found:", best_path)
    print("Best Path Length:", best_length)
```

```
[Running] python -u "c:\Users\BMSCE\Documents\1BM23CS159\BIS\LAB3\TSP.py"
Iteration 1/20 - Best Length: 43.00
Iteration 2/20 - Best Length: 43.00
Iteration 3/20 - Best Length: 43.00
Iteration 4/20 - Best Length: 43.00
Iteration 5/20 - Best Length: 43.00
Iteration 6/20 - Best Length: 43.00
Iteration 7/20 - Best Length: 43.00
Iteration 8/20 - Best Length: 43.00
Iteration 9/20 - Best Length: 43.00
Iteration 10/20 - Best Length: 43.00
Iteration 11/20 - Best Length: 43.00
Iteration 12/20 - Best Length: 43.00
Iteration 13/20 - Best Length: 43.00
Iteration 14/20 - Best Length: 43.00
Iteration 15/20 - Best Length: 43.00
Iteration 16/20 - Best Length: 43.00
Iteration 17/20 - Best Length: 43.00
Iteration 18/20 - Best Length: 43.00
Iteration 19/20 - Best Length: 43.00
Iteration 20/20 - Best Length: 43.00

Best Path Found: [1, 0, 3, 2]
Best Path Length: 43

[Done] exited with code=0 in 0.123 seconds
```

Program 4:
Cuckoo Search (CS)

Cuckoo Search Algorithm

Input:

objective function ($f(x)$)

search bound

no. of nest (n)

abanding probability (pa)

~~best~~ & Maximum No of Iteration (MaxGen)

Output: Best nest(x_{best}) and bestsolution ($f(x_{best})$)

1. For all n nest randomly initialize x_i

2. Compute fitness of all nest

3. Assign the best fitness value for best nest

$$x_{best} = x_i(\text{best fitness})$$

4. For $t=1$ to MaxGen

a. For a random nest in search bound calculate the value using Levy flight

$$x_{new} = x_i + \alpha \, Levy(\lambda)$$

b. For a random nest j in nest

if $f(x_j) \geq f(x_{best})$

Replace x_j with x_{best}

$$x_j = x_{best}$$

c. Abandon the worst nest with fraction (pa) and replace them with new random value

d. Evaluate the fitness of all.

e. Replace xbest, if best solution is found

(End for)

6. Return xbest and f(xbest)

```
import numpy as np
import math

# ------------------------------
# Step 1: Example financial data
# ------------------------------
# Expected returns of 4 assets
returns = np.array([0.12, 0.10, 0.15, 0.09])

# Covariance matrix of asset returns (risk relationships)
cov_matrix = np.array([
    [0.010, 0.002, 0.001, 0.003],
    [0.002, 0.008, 0.002, 0.002],
    [0.001, 0.002, 0.012, 0.004],
    [0.003, 0.002, 0.004, 0.009]
])

num_assets = len(returns)

# ------------------------------
# Step 2: Fitness function
# ------------------------------
def portfolio_fitness(weights, alpha=0.5, beta=0.5):
    weights = np.array(weights)
    weights = np.clip(weights, 0, 1)  # bounds [0,1]
    weights /= np.sum(weights)        # normalize (budget constraint)

    expected_return = np.dot(weights, returns)
    risk = np.dot(weights.T, np.dot(cov_matrix, weights))

    # lower fitness = better (we minimize risk - return)
    fitness = alpha * risk - beta * expected_return
```

```
    return fitness, expected_return, risk

# ------------------------------
# Step 3: Cuckoo Search Algorithm
# ------------------------------
def levy_flight(Lambda):
    sigma = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
             (math.gamma((1 + Lambda)/2) * Lambda * 2**((Lambda-1)/2)))**(1/Lambda)
    u = np.random.normal(0, sigma, num_assets)
    v = np.random.normal(0, 1, num_assets)
    step = u / np.abs(v)**(1/Lambda)
    return step

def cuckoo_search(n=20, max_iter=100, pa=0.25):
    nests = np.random.dirichlet(np.ones(num_assets), size=n)
    fitness = [portfolio_fitness(w)[0] for w in nests]
    best_idx = np.argmin(fitness)
    best = nests[best_idx]

    for _ in range(max_iter):
        for i in range(n):
            step_size = levy_flight(1.5)
            new_solution = nests[i] + step_size * np.random.randn(num_assets)
            new_solution = np.clip(new_solution, 0, 1)
            new_solution /= np.sum(new_solution)

            new_fitness = portfolio_fitness(new_solution)[0]
            if new_fitness < fitness[i]:
                nests[i] = new_solution
                fitness[i] = new_fitness

        # Abandon some nests with probability pa
        for i in range(n):
            if np.random.rand() < pa:
                nests[i] = np.random.dirichlet(np.ones(num_assets))
                fitness[i] = portfolio_fitness(nests[i])[0]

        # Update best nest
        best_idx = np.argmin(fitness)
        best = nests[best_idx]

    return best, portfolio_fitness(best)

# ------------------------------
# Step 4: Run optimization
# ------------------------------
best_weights, (fitness_value, best_return, best_risk) = cuckoo_search()
```

```
print("Optimal Portfolio Allocation:")
for i, w in enumerate(best_weights):
    print(f"  Asset {i+1}: {w:.2f}")

print(f"\nExpected Return: {best_return:.4f}")
print(f"Risk (Variance): {best_risk:.4f}")
```

```
[Running] python -u "c:\Users\BMSCE\Documents\1BM23CS159\BIS\LAB4\CSA(Finance Portfolio Optimisation).py"
c:\Users\BMSCE\Documents\1BM23CS159\BIS\LAB4\CSA(Finance Portfolio Optimisation).py:57: RuntimeWarning: invalid value encountered in divide
  new_solution /= np.sum(new_solution)
Optimal Portfolio Allocation:
  Asset 1: 0.00
  Asset 2: 0.00
  Asset 3: 1.00
  Asset 4: 0.00

Expected Return: 0.1500
Risk (Variance): 0.0120

[Done] exited with code=0 in 0.217 seconds
```

Program 5:

Grey Wolf Optimizer (GWO):

Bafna Gold

Date: Page:

Grey Wolf Optimisation

Input :

Objective function $f(x)$

Maximum Iteration (MaxItr)

Alpha (α)

Beta (β)

~~Omega~~ Delta

Lowerbound & upper bound (lo, ua)

Output :

Alpa wolf (α)

fitness of Alpa wolf($f(\alpha)$)

1. Initialize the wolf (parameters) population

for each wolf i

position(i) = Random value between lowerbound & upperbound

2. Evaluate the fitness

& fitness (i) = f(position(i))

3. Update position.

alpha_score = 10000

Beta_score = 10000

Delta_score = 10000

4. for each iteration t=1 to MaxIteration

for each wolf i=1 to population size:

a. ensure x_i is within la & ua

b. calculate fitness = $f(x_i)$

c. if fitness < alpha_score:

delta_score = beta_score

~~d. if fitness < del~~

delta-pos = beta-pos;

beta-score = alpha_score ;

beta pos = alpha score
alpha score = fitness
alpha pos = x_i

else if fitness < beta score:
delta score = beta score
delta pos = beta pos
beta score = fitness
beta pos = x_i
else if fitness < delta score:
delta score = fitness
delta pos = x_i

(End for)

4. Compute parameter a = 2 - 2 (2 * t / max iteration)

5. For each wolf i = 1 to N
for each dimension j = 1 to population size:
Generate random number r_1, r_2, ∈ [0, 1]

compute:
A1 = 2 * a * r - a
C1 = 2 * r_2
D alpha = [C1 * Alpha_pos[j] - x_i[j]]
X1 = Alpha pos [j] - A1 * D-alpha

Repeat similarly for Beta & Delta
A2, C2, D_beta, X2
A3, C3, D_Delta, X3

Update the position

$x_i[j]$ = (X1 + X2 + X3) / 3

(End for)

6. Return Alpha pos & fitness (Alpha_pos)

image enhancement

```
import numpy as np
import cv2
from skimage.metrics import structural_similarity as ssim
from skimage.metrics import peak_signal_noise_ratio as psnr
from scipy.stats import entropy
import matplotlib.pyplot as plt
from typing import Tuple, List
import warnings
warnings.filterwarnings('ignore')


class GreyWolfOptimizer:
    """Grey Wolf Optimizer for image enhancement parameters"""

    def __init__(self, n_wolves=10, max_iter=30, dim=5):
        self.n_wolves = n_wolves
        self.max_iter = max_iter
        self.dim = dim

        # Parameter bounds: [alpha, beta, R, G, B]
        self.lb = np.array([0.5, 0.5, 0.5, 0.5, 0.5])
        self.ub = np.array([2.0, 2.0, 1.5, 1.5, 1.5])

        # Initialize wolf positions
        self.positions = np.random.uniform(
            self.lb, self.ub, (self.n_wolves, self.dim)
        )

        # Alpha, Beta, Delta wolves (best solutions)
        self.alpha_pos = np.zeros(self.dim)
        self.alpha_score = float('-inf')

        self.beta_pos = np.zeros(self.dim)
        self.beta_score = float('-inf')

        self.delta_pos = np.zeros(self.dim)
        self.delta_score = float('-inf')

        self.convergence_curve = []

    def optimize(self, fitness_func):
        """Run GWO optimization"""
        print("Starting Grey Wolf Optimization...")

        for iteration in range(self.max_iter):
            # Evaluate fitness for all wolves
            for i in range(self.n_wolves):
```

```
        fitness = fitness_func(self.positions[i])

        # Update Alpha, Beta, Delta
        if fitness > self.alpha_score:
            self.delta_score = self.beta_score
            self.delta_pos = self.beta_pos.copy()

            self.beta_score = self.alpha_score
            self.beta_pos = self.alpha_pos.copy()

            self.alpha_score = fitness
            self.alpha_pos = self.positions[i].copy()

        elif fitness > self.beta_score:
            self.delta_score = self.beta_score
            self.delta_pos = self.beta_pos.copy()

            self.beta_score = fitness
            self.beta_pos = self.positions[i].copy()

        elif fitness > self.delta_score:
            self.delta_score = fitness
            self.delta_pos = self.positions[i].copy()

    # Linearly decrease 'a' from 2 to 0
    a = 2 - iteration * (2 / self.max_iter)

    # Update positions of all wolves
    for i in range(self.n_wolves):
        for j in range(self.dim):
            # Update using Alpha
            r1, r2 = np.random.random(2)
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * self.alpha_pos[j] - self.positions[i, j])
            X1 = self.alpha_pos[j] - A1 * D_alpha

            # Update using Beta
            r1, r2 = np.random.random(2)
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * self.beta_pos[j] - self.positions[i, j])
            X2 = self.beta_pos[j] - A2 * D_beta

            # Update using Delta
            r1, r2 = np.random.random(2)
            A3 = 2 * a * r1 - a
```

```
                C3 = 2 * r2
                D_delta = abs(C3 * self.delta_pos[j] - self.positions[i, j])
                X3 = self.delta_pos[j] - A3 * D_delta

                # Average position
                self.positions[i, j] = (X1 + X2 + X3) / 3

                # Boundary check
                self.positions[i, j] = np.clip(
                    self.positions[i, j], self.lb[j], self.ub[j]
                )

            self.convergence_curve.append(self.alpha_score)

            if (iteration + 1) % 5 == 0:
                print(f"Iteration {iteration + 1}/{self.max_iter}, "
                      f"Best Fitness: {self.alpha_score:.4f}")

        print(f"\nOptimization Complete!")
        print(f"Best Parameters: α={self.alpha_pos[0]:.3f}, "
              f"β={self.alpha_pos[1]:.3f}, "
              f"R={self.alpha_pos[2]:.3f}, "
              f"G={self.alpha_pos[3]:.3f}, "
              f"B={self.alpha_pos[4]:.3f}")

        return self.alpha_pos, self.alpha_score


class ImageEnhancer:
    """Automatic Image Enhancement System"""

    def __init__(self, image_path: str):
        self.original_image = cv2.imread(image_path)
        if self.original_image is None:
            raise ValueError(f"Could not read image from {image_path}")

        self.original_image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2RGB)
        self.enhanced_image = None
        self.best_params = None

        print(f"Image loaded: {self.original_image.shape}")

    def apply_enhancement(self, params: np.ndarray, image: np.ndarray = None) -> np.ndarray:
        """Apply enhancement parameters to image"""
        if image is None:
            image = self.original_image
```

```
        alpha, beta, R, G, B = params

        # Convert to float for processing
        enhanced = image.astype(np.float32)

        # Adjust brightness and contrast
        enhanced = cv2.convertScaleAbs(enhanced, alpha=alpha, beta=(beta - 1) * 50)
        enhanced = enhanced.astype(np.float32)

        # Adjust color balance for each channel
        enhanced[:, :, 0] = np.clip(enhanced[:, :, 0] * R, 0, 255)  # R
        enhanced[:, :, 1] = np.clip(enhanced[:, :, 1] * G, 0, 255)  # G
        enhanced[:, :, 2] = np.clip(enhanced[:, :, 2] * B, 0, 255)  # B

        return enhanced.astype(np.uint8)

    def calculate_entropy(self, image: np.ndarray) -> float:
        """Calculate image entropy (information content)"""
        gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
        hist, _ = np.histogram(gray, bins=256, range=(0, 256))
        hist = hist / hist.sum()
        hist = hist[hist > 0]  # Remove zero probabilities
        return entropy(hist, base=2)

    def calculate_edge_intensity(self, image: np.ndarray) -> float:
        """Calculate average edge intensity using Sobel operator"""
        gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

        # Sobel edge detection
        sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
        sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)

        edge_magnitude = np.sqrt(sobelx**2 + sobely**2)
        return np.mean(edge_magnitude)

    def calculate_color_contrast(self, image: np.ndarray) -> float:
        """Calculate color contrast using standard deviation"""
        std_r = np.std(image[:, :, 0])
        std_g = np.std(image[:, :, 1])
        std_b = np.std(image[:, :, 2])
        return (std_r + std_g + std_b) / 3

    def fitness_function(self, params: np.ndarray) -> float:
        """
        Fitness function combining entropy, edge intensity, and color contrast
        Higher values indicate better image quality
        """
```

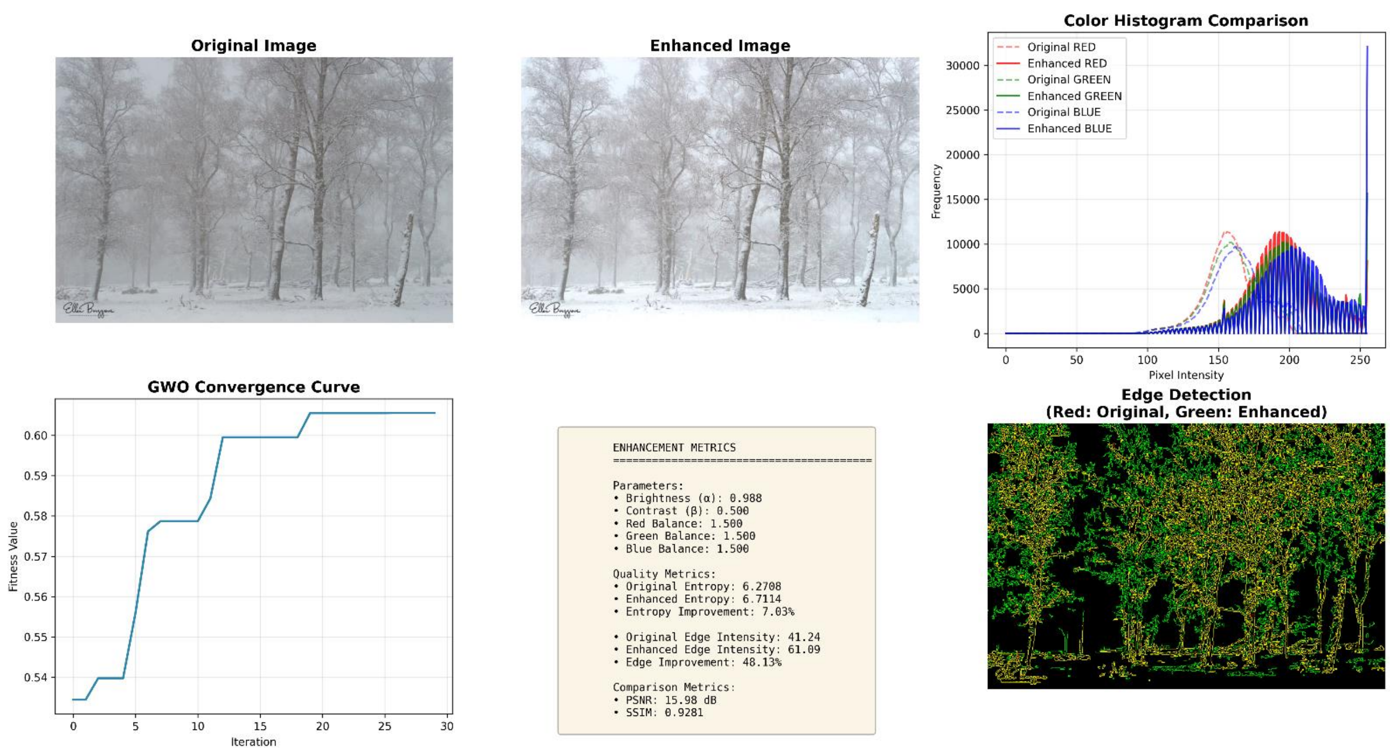
```
    try:
        enhanced = self.apply_enhancement(params)

        # Calculate quality metrics
        entropy_val = self.calculate_entropy(enhanced)
        edge_intensity = self.calculate_edge_intensity(enhanced)
        color_contrast = self.calculate_color_contrast(enhanced)

        # Normalize and combine metrics
        # Weights can be adjusted based on importance
        w1, w2, w3 = 0.4, 0.3, 0.3

        fitness = (w1 * entropy_val / 8.0 +  # Normalize entropy (max ~8)
                  w2 * edge_intensity / 100.0 +  # Normalize edge intensity
                  w3 * color_contrast / 100.0)  # Normalize color contrast

        return fitness

    except Exception as e:
        return float('-inf')

def enhance(self, n_wolves=10, max_iter=30):
    """Perform automatic enhancement using GWO"""
    print("\n" + "="*60)
    print("AUTOMATIC IMAGE ENHANCEMENT USING GREY WOLF OPTIMIZER")
    print("="*60 + "\n")

    # Initialize GWO
    gwo = GreyWolfOptimizer(n_wolves=n_wolves, max_iter=max_iter, dim=5)

    # Run optimization
    self.best_params, best_fitness = gwo.optimize(self.fitness_function)

    # Apply best parameters
    self.enhanced_image = self.apply_enhancement(self.best_params)

    return self.enhanced_image, self.best_params, gwo.convergence_curve

def calculate_metrics(self) -> dict:
    """Calculate comparison metrics between original and enhanced images"""
    if self.enhanced_image is None:
        raise ValueError("No enhanced image available. Run enhance() first.")

    # Convert to grayscale for some metrics
    orig_gray = cv2.cvtColor(self.original_image, cv2.COLOR_RGB2GRAY)
    enh_gray = cv2.cvtColor(self.enhanced_image, cv2.COLOR_RGB2GRAY)
```

```
        metrics = {
            'original_entropy': self.calculate_entropy(self.original_image),
            'enhanced_entropy': self.calculate_entropy(self.enhanced_image),
            'original_edge_intensity': self.calculate_edge_intensity(self.original_image),
            'enhanced_edge_intensity': self.calculate_edge_intensity(self.enhanced_image),
            'psnr': psnr(self.original_image, self.enhanced_image),
            'ssim': ssim(orig_gray, enh_gray),
        }

        metrics['entropy_improvement'] = (
            (metrics['enhanced_entropy'] - metrics['original_entropy']) /
            metrics['original_entropy'] * 100
        )

        metrics['edge_improvement'] = (
            (metrics['enhanced_edge_intensity'] - metrics['original_edge_intensity']) /
            metrics['original_edge_intensity'] * 100
        )

        return metrics

    def visualize_results(self, convergence_curve: List[float], metrics: dict):
        """Visualize original, enhanced images and metrics"""
        fig = plt.figure(figsize=(18, 10))

        # Original Image
        ax1 = plt.subplot(2, 3, 1)
        ax1.imshow(self.original_image)
        ax1.set_title('Original Image', fontsize=14, fontweight='bold')
        ax1.axis('off')

        # Enhanced Image
        ax2 = plt.subplot(2, 3, 2)
        ax2.imshow(self.enhanced_image)
        ax2.set_title('Enhanced Image', fontsize=14, fontweight='bold')
        ax2.axis('off')

        # Histogram Comparison
        ax3 = plt.subplot(2, 3, 3)
        for i, color in enumerate(['red', 'green', 'blue']):
            hist_orig, _ = np.histogram(self.original_image[:, :, i], bins=256, range=(0, 256))
            hist_enh, _ = np.histogram(self.enhanced_image[:, :, i], bins=256, range=(0, 256))
            ax3.plot(hist_orig, color=color, alpha=0.5, linestyle='--', label=f'Original {color.upper()}')
            ax3.plot(hist_enh, color=color, alpha=0.8, label=f'Enhanced {color.upper()}')
        ax3.set_title('Color Histogram Comparison', fontsize=14, fontweight='bold')
        ax3.set_xlabel('Pixel Intensity')
        ax3.set_ylabel('Frequency')
```

```
ax3.legend()
ax3.grid(True, alpha=0.3)

# Convergence Curve
ax4 = plt.subplot(2, 3, 4)
ax4.plot(convergence_curve, linewidth=2, color='#2E86AB')
ax4.set_title('GWO Convergence Curve', fontsize=14, fontweight='bold')
ax4.set_xlabel('Iteration')
ax4.set_ylabel('Fitness Value')
ax4.grid(True, alpha=0.3)

# Metrics Table
ax5 = plt.subplot(2, 3, 5)
ax5.axis('off')

metrics_text = f"""
ENHANCEMENT METRICS
{'='*40}

Parameters:
• Brightness (α): {self.best_params[0]:.3f}
• Contrast (β): {self.best_params[1]:.3f}
• Red Balance: {self.best_params[2]:.3f}
• Green Balance: {self.best_params[3]:.3f}
• Blue Balance: {self.best_params[4]:.3f}

Quality Metrics:
• Original Entropy: {metrics['original_entropy']:.4f}
• Enhanced Entropy: {metrics['enhanced_entropy']:.4f}
• Entropy Improvement: {metrics['entropy_improvement']:.2f}%

• Original Edge Intensity: {metrics['original_edge_intensity']:.2f}
• Enhanced Edge Intensity: {metrics['enhanced_edge_intensity']:.2f}
• Edge Improvement: {metrics['edge_improvement']:.2f}%

Comparison Metrics:
• PSNR: {metrics['psnr']:.2f} dB
• SSIM: {metrics['ssim']:.4f}
"""

ax5.text(0.1, 0.9, metrics_text, transform=ax5.transAxes,
        fontsize=10, verticalalignment='top', fontfamily='monospace',
        bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.3))

# Edge Detection Comparison
ax6 = plt.subplot(2, 3, 6)
orig_gray = cv2.cvtColor(self.original_image, cv2.COLOR_RGB2GRAY)
```

```python
        enh_gray = cv2.cvtColor(self.enhanced_image, cv2.COLOR_RGB2GRAY)

        edges_orig = cv2.Canny(orig_gray, 50, 150)
        edges_enh = cv2.Canny(enh_gray, 50, 150)

        edges_combined = np.zeros((*edges_orig.shape, 3), dtype=np.uint8)
        edges_combined[:, :, 0] = edges_orig  # Red channel - original
        edges_combined[:, :, 1] = edges_enh   # Green channel - enhanced

        ax6.imshow(edges_combined)
        ax6.set_title('Edge Detection\n(Red: Original, Green: Enhanced)',
                      fontsize=14, fontweight='bold')
        ax6.axis('off')

        plt.tight_layout()
        plt.savefig('enhancement_results.png', dpi=300, bbox_inches='tight')
        print("\nResults saved to 'enhancement_results.png'")
        plt.show()


def main():
    """Main function to run the image enhancement system"""

    # Example usage
    image_path = 'input_image.jpg'  # Replace with your image path

    try:
        # Create enhancer instance
        enhancer = ImageEnhancer(image_path)

        # Perform enhancement
        enhanced_img, best_params, convergence = enhancer.enhance(
            n_wolves=15,  # Number of wolves (search agents)
            max_iter=30   # Number of iterations
        )

        # Calculate metrics
        metrics = enhancer.calculate_metrics()

        # Visualize results
        enhancer.visualize_results(convergence, metrics)

        # Save enhanced image
        cv2.imwrite('enhanced_image.jpg',
                    cv2.cvtColor(enhanced_img, cv2.COLOR_RGB2BGR))
        print("\nEnhanced image saved to 'enhanced_image.jpg'")
```

```
        # Print summary
        print("\n" + "="*60)
        print("ENHANCEMENT SUMMARY")
        print("="*60)
        print(f"Entropy Improvement: {metrics['entropy_improvement']:.2f}%")
        print(f"Edge Intensity Improvement: {metrics['edge_improvement']:.2f}%")
        print(f"PSNR: {metrics['psnr']:.2f} dB")
        print(f"SSIM: {metrics['ssim']:.4f}")
        print("="*60)

    except Exception as e:
        print(f"Error: {str(e)}")
        print("\nPlease ensure you have an image file named 'input_image.jpg'")
        print("or modify the image_path variable in the main() function.")


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



Program 6:

Parallel Cellular Algorithms and Programs:

Bafna Gold

Date: Page:

Parallel Cellular Optimisation

Input :

f(x) : function to optimize

numcells : number of cells

max_iter : maximum iteration

grid_size : size of grid in m×n

neighborhood

Output :

best position

~~best solution~~ best fitness : f(best_position)

1. Initialize parameters

   Input : f(x), numcells, max_iter, gridsize

2. Initialize population

   for each cell i in grid :

   cell[i].position ← random position in solution space

   cell[i].fitness ← f(cell[i].position)

3. Identify initial best solution

   best solution ← cell with minimum fitness

   best fitness ← f(best_solution)

4. for iter = 1 to max_iter do :

   In parallel for each cell i :

   • get neighbors N_i according to neighborhood structure

   • Compute newstate ← update_rule(cell[i], N_i)

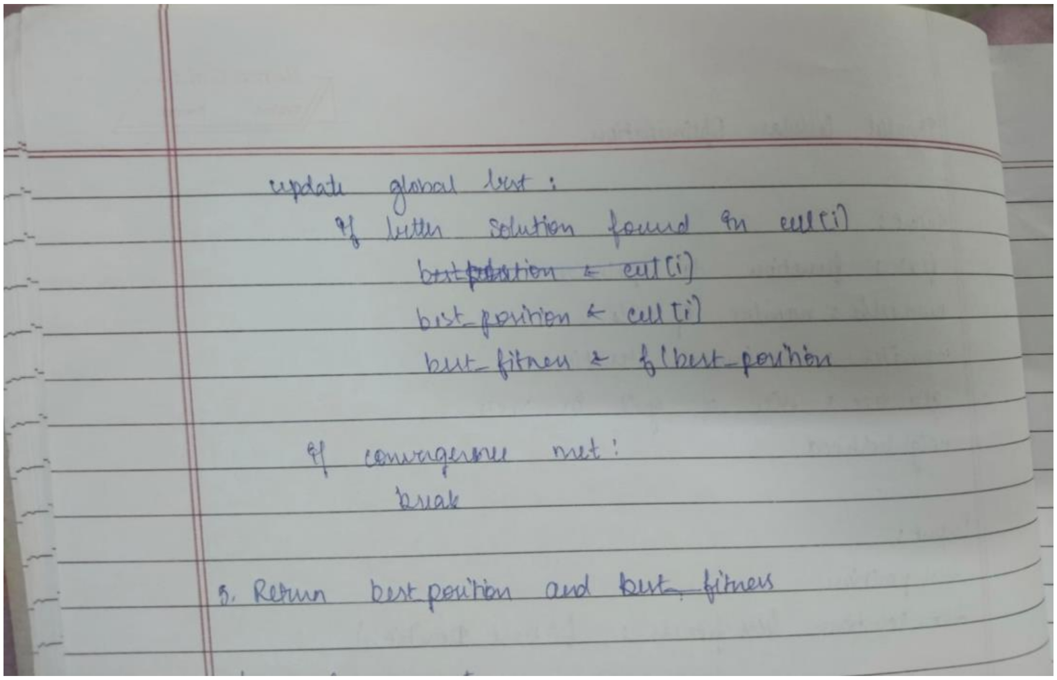
   • Evaluate new_fitness ← f(newstate)

   • If new_fitness better than cell[i]

   cell[i].position ← ~~f(~~ newstate

   cell[i].fitness ← f(new_state)

   [End parallel]


update global best :
if better solution found in cell(i)
~~best position ← cell(i)~~
best_position ← cell(i)
best_fitness ← f(best_position
if convergence met :
break
5. Return best_position and best_fitness

```
import numpy as np
import cv2
from scipy.ndimage import sobel
import matplotlib.pyplot as plt
import os

class ParallelCellularAlgorithm:
    """
    Parallel Cellular Algorithm for Image Enhancement
    Optimizes brightness and contrast parameters
    """

    def __init__(self, image_path, grid_size=10, max_iterations=200):
        # Load image
        if not os.path.exists(image_path):
            raise FileNotFoundError(f"Image file not found: {image_path}")

        self.original_image = cv2.imread(image_path)
        if self.original_image is None:
            raise ValueError(f"Could not load image from {image_path}")

        print(f"Loaded image: {self.original_image.shape}")

        # Convert to grayscale for processing
        self.gray_image = cv2.cvtColor(self.original_image, cv2.COLOR_BGR2GRAY)

        # PCA Parameters
        self.grid_size = grid_size
```

```
        self.total_cells = grid_size * grid_size
        self.max_iterations = max_iterations

        # Parameter ranges
        self.B_MIN, self.B_MAX = -50, 50  # Brightness range
        self.C_MIN, self.C_MAX = 0.5, 2.0  # Contrast range

        # Fitness weights
        self.w1 = 0.4  # Entropy weight
        self.w2 = 0.4  # Edge density weight
        self.w3 = 0.2  # MSE penalty weight

        # Initialize grid
        self.grid = np.zeros((grid_size, grid_size), dtype=object)
        self.initialize_grid()

        # Tracking
        self.best_B = 0
        self.best_C = 1.0
        self.best_fitness = -float('inf')
        self.fitness_history = []
        self.best_params_history = []

    def initialize_grid(self):
        """Initialize each cell with random B and C values"""
        for i in range(self.grid_size):
            for j in range(self.grid_size):
                B = np.random.uniform(self.B_MIN, self.B_MAX)
                C = np.random.uniform(self.C_MIN, self.C_MAX)
                self.grid[i, j] = {'B': B, 'C': C, 'fitness': 0}

    def apply_enhancement(self, image, B, C):
        """
        Apply brightness and contrast adjustment
        Formula: output = C * (input - 128) + 128 + B
        """
        # Convert to float for precision
        enhanced = image.astype(np.float32)

        # Apply contrast around midpoint (128) then add brightness
        enhanced = C * (enhanced - 128.0) + 128.0 + B

        # Clip to valid range [0, 255]
        enhanced = np.clip(enhanced, 0, 255).astype(np.uint8)

        return enhanced
```

```
def calculate_entropy(self, image):
    """Calculate Shannon entropy of the image"""
    histogram, _ = np.histogram(image.flatten(), bins=256, range=(0, 256))
    histogram = histogram / histogram.sum()  # Normalize

    # Remove zeros to avoid log(0)
    histogram = histogram[histogram > 0]

    # Shannon entropy
    entropy = -np.sum(histogram * np.log2(histogram))
    return entropy

def calculate_edge_density(self, image):
    """Calculate edge density using Sobel operator"""
    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(image, (3, 3), 0)

    # Compute gradients using Sobel
    grad_x = cv2.Sobel(blurred, cv2.CV_64F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(blurred, cv2.CV_64F, 0, 1, ksize=3)

    # Magnitude of gradient
    gradient_magnitude = np.sqrt(grad_x**2 + grad_y**2)

    # Edge density (normalized)
    edge_density = np.mean(gradient_magnitude) / 255.0
    return edge_density

def calculate_mse(self, original, enhanced):
    """Calculate Mean Squared Error (normalized)"""
    mse = np.mean((original.astype(np.float32) - enhanced.astype(np.float32)) ** 2)
    return mse / (255.0 * 255.0)  # Normalize to [0, 1]

def fitness_function(self, B, C):
    """
    Fitness function to evaluate image quality
    F(B,C) = w1*Entropy + w2*EdgeDensity - w3*MSE
    """
    # Apply enhancement to grayscale image
    enhanced = self.apply_enhancement(self.gray_image, B, C)

    # Calculate metrics
    entropy = self.calculate_entropy(enhanced)
    edge_density = self.calculate_edge_density(enhanced)
    mse = self.calculate_mse(self.gray_image, enhanced)

    # Compute fitness (higher is better)
```

```
        fitness = self.w1 * entropy + self.w2 * edge_density - self.w3 * mse

        return fitness

    def get_moore_neighbors(self, i, j):
        """Get Moore neighborhood (8 neighbors) with toroidal wrap-around"""
        neighbors = []
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if di == 0 and dj == 0:
                    continue
                ni, nj = i + di, j + dj
                # Wrap around (toroidal topology)
                ni = ni % self.grid_size
                nj = nj % self.grid_size
                neighbors.append((ni, nj))
        return neighbors

    def update_cell(self, i, j):
        """Update cell based on best neighbor using diffusion rule"""
        current_cell = self.grid[i, j]
        neighbors = self.get_moore_neighbors(i, j)

        # Find best neighbor
        best_neighbor_fitness = current_cell['fitness']
        best_neighbor = current_cell

        for ni, nj in neighbors:
            neighbor = self.grid[ni, nj]
            if neighbor['fitness'] > best_neighbor_fitness:
                best_neighbor_fitness = neighbor['fitness']
                best_neighbor = neighbor

        # Update using diffusion rule (move toward best neighbor)
        alpha = 0.3  # Learning rate
        new_B = current_cell['B'] + alpha * (best_neighbor['B'] - current_cell['B'])
        new_C = current_cell['C'] + alpha * (best_neighbor['C'] - current_cell['C'])

        # Add small random exploration
        new_B += np.random.normal(0, 2)
        new_C += np.random.normal(0, 0.05)

        # Clip to valid ranges
        new_B = np.clip(new_B, self.B_MIN, self.B_MAX)
        new_C = np.clip(new_C, self.C_MIN, self.C_MAX)

        return {'B': new_B, 'C': new_C, 'fitness': 0}
```

```
def evaluate_grid(self):
    """Evaluate fitness for all cells in parallel"""
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            cell = self.grid[i, j]
            cell['fitness'] = self.fitness_function(cell['B'], cell['C'])

            # Track global best
            if cell['fitness'] > self.best_fitness:
                self.best_fitness = cell['fitness']
                self.best_B = cell['B']
                self.best_C = cell['C']

def run(self, verbose=True):
    """Run the PCA optimization"""
    if verbose:
        print("\n" + "="*60)
        print("Starting Parallel Cellular Algorithm")
        print("="*60)
        print(f"Grid Size: {self.grid_size}x{self.grid_size} ({self.total_cells} cells)")
        print(f"Max Iterations: {self.max_iterations}")
        print(f"Brightness Range: [{self.B_MIN}, {self.B_MAX}]")
        print(f"Contrast Range: [{self.C_MIN}, {self.C_MAX}]")
        print("-" * 60)

    import time
    start_time = time.time()

    for iteration in range(self.max_iterations):
        # Evaluate all cells
        self.evaluate_grid()

        # Store history
        self.fitness_history.append(self.best_fitness)
        self.best_params_history.append((self.best_B, self.best_C))

        # Update all cells (parallel update - synchronous)
        new_grid = np.zeros((self.grid_size, self.grid_size), dtype=object)
        for i in range(self.grid_size):
            for j in range(self.grid_size):
                new_grid[i, j] = self.update_cell(i, j)

        self.grid = new_grid

        # Print progress
        if verbose and (iteration + 1) % 20 == 0:
```

```
            print(f"Iteration {iteration + 1:3d}/{self.max_iterations} | "
                  f"Fitness: {self.best_fitness:7.4f} | "
                  f"B: {self.best_B:6.2f} | C: {self.best_C:5.3f}")

    end_time = time.time()

    if verbose:
        print("-" * 60)
        print(f"Optimization Complete!")
        print(f"Time: {end_time - start_time:.2f} seconds")
        print(f"Best Fitness: {self.best_fitness:.4f}")
        print(f"Optimal B*: {self.best_B:.2f}")
        print(f"Optimal C*: {self.best_C:.3f}")
        print("="*60 + "\n")

    return self.best_B, self.best_C

def get_enhanced_image(self, B=None, C=None):
    """
    Get enhanced image using specified or optimal parameters
    Applies enhancement to full color image
    """
    if B is None:
        B = self.best_B
    if C is None:
        C = self.best_C

    # Apply to each channel of the color image
    enhanced_color = self.original_image.copy()

    for channel in range(3):
        enhanced_color[:, :, channel] = self.apply_enhancement(
            self.original_image[:, :, channel], B, C
        )

    return enhanced_color

def visualize_results(self, save_path=None):
    """Visualize original, enhanced images and convergence"""
    fig = plt.figure(figsize=(16, 10))

    # Create grid layout
    gs = fig.add_gridspec(3, 3, hspace=0.3, wspace=0.3)

    # Original image
    ax1 = fig.add_subplot(gs[0:2, 0])
    ax1.imshow(cv2.cvtColor(self.original_image, cv2.COLOR_BGR2RGB))
```

```
ax1.set_title('Original Image', fontsize=14, fontweight='bold', pad=10)
ax1.axis('off')

# Enhanced image
ax2 = fig.add_subplot(gs[0:2, 1])
enhanced = self.get_enhanced_image()
ax2.imshow(cv2.cvtColor(enhanced, cv2.COLOR_BGR2RGB))
ax2.set_title(f'Enhanced Image\nB* = {self.best_B:.2f}, C* = {self.best_C:.3f}',
        fontsize=14, fontweight='bold', pad=10)
ax2.axis('off')

# Histograms comparison
ax3 = fig.add_subplot(gs[0:2, 2])
ax3.hist(self.gray_image.flatten(), bins=256, range=(0, 256),
     alpha=0.5, color='blue', label='Original', density=True)
enhanced_gray = self.apply_enhancement(self.gray_image, self.best_B, self.best_C)
ax3.hist(enhanced_gray.flatten(), bins=256, range=(0, 256),
     alpha=0.5, color='red', label='Enhanced', density=True)
ax3.set_xlabel('Pixel Intensity', fontsize=11)
ax3.set_ylabel('Density', fontsize=11)
ax3.set_title('Histogram Comparison', fontsize=12, fontweight='bold')
ax3.legend()
ax3.grid(True, alpha=0.3)

# Fitness convergence
ax4 = fig.add_subplot(gs[2, 0])
ax4.plot(self.fitness_history, linewidth=2, color='#2E86AB')
ax4.set_xlabel('Iteration', fontsize=11)
ax4.set_ylabel('Best Fitness', fontsize=11)
ax4.set_title('Fitness Convergence', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)

# Parameter evolution
ax5 = fig.add_subplot(gs[2, 1])
B_history = [p[0] for p in self.best_params_history]
C_history = [p[1] for p in self.best_params_history]

ax5_twin = ax5.twinx()

line1 = ax5.plot(B_history, linewidth=2, color='#A23B72', label='Brightness')
ax5.set_xlabel('Iteration', fontsize=11)
ax5.set_ylabel('Brightness (B)', fontsize=11, color='#A23B72')
ax5.tick_params(axis='y', labelcolor='#A23B72')

line2 = ax5_twin.plot(C_history, linewidth=2, color='#F18F01', label='Contrast')
ax5_twin.set_ylabel('Contrast (C)', fontsize=11, color='#F18F01')
ax5_twin.tick_params(axis='y', labelcolor='#F18F01')
```

```
        ax5.set_title('Parameter Evolution', fontsize=12, fontweight='bold')
        ax5.grid(True, alpha=0.3)

        # Metrics comparison
        ax6 = fig.add_subplot(gs[2, 2])

        # Calculate metrics for both images
        orig_entropy = self.calculate_entropy(self.gray_image)
        orig_edge = self.calculate_edge_density(self.gray_image)

        enh_entropy = self.calculate_entropy(enhanced_gray)
        enh_edge = self.calculate_edge_density(enhanced_gray)

        metrics = ['Entropy', 'Edge\nDensity']
        orig_vals = [orig_entropy/8, orig_edge]  # Normalize entropy for display
        enh_vals = [enh_entropy/8, enh_edge]

        x = np.arange(len(metrics))
        width = 0.35

        ax6.bar(x - width/2, orig_vals, width, label='Original', color='#4A90E2')
        ax6.bar(x + width/2, enh_vals, width, label='Enhanced', color='#E94B3C')

        ax6.set_ylabel('Normalized Value', fontsize=11)
        ax6.set_title('Quality Metrics', fontsize=12, fontweight='bold')
        ax6.set_xticks(x)
        ax6.set_xticklabels(metrics, fontsize=10)
        ax6.legend()
        ax6.grid(True, alpha=0.3, axis='y')

        plt.suptitle('Image Enhancement using Parallel Cellular Algorithm',
                     fontsize=16, fontweight='bold', y=0.98)

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')
            print(f"Results saved to: {save_path}")

        plt.show()

    def save_enhanced_image(self, output_path):
        """Save the enhanced image"""
        enhanced = self.get_enhanced_image()
        cv2.imwrite(output_path, enhanced)
        print(f"Enhanced image saved to: {output_path}")
```

```
# Example usage
if __name__ == "__main__":
    # IMPORTANT: Replace 'your_image.jpg' with your actual image path
    image_path = 'download.jpg'  # <-- CHANGE THIS

    # Check if image exists
    if not os.path.exists(image_path):
        print(f"ERROR: Image file '{image_path}' not found!")
        print("Please provide a valid image path.")
        print("\nCreating a sample image for demonstration...")

        # Create a more realistic sample image (dark, low contrast)
        sample = np.ones((400, 600, 3), dtype=np.uint8) * 100
        # Add some structure
        sample[50:150, 50:250] = 140
        sample[200:350, 300:550] = 120
        sample = cv2.GaussianBlur(sample, (15, 15), 0)

        image_path = 'sample_low_contrast.jpg'
        cv2.imwrite(image_path, sample)
        print(f"Sample image created: {image_path}")

    try:
        # Initialize PCA
        pca = ParallelCellularAlgorithm(
            image_path=image_path,
            grid_size=10,
            max_iterations=200
        )

        # Run optimization
        optimal_B, optimal_C = pca.run(verbose=True)

        # Visualize results
        pca.visualize_results(save_path='enhancement_results.png')

        # Save enhanced image
        pca.save_enhanced_image('enhanced_image.jpg')

        print("\nAll outputs saved successfully!")

    except Exception as e:
        print(f"\nError: {e}")
        print("Please check your image path and try again.")
```

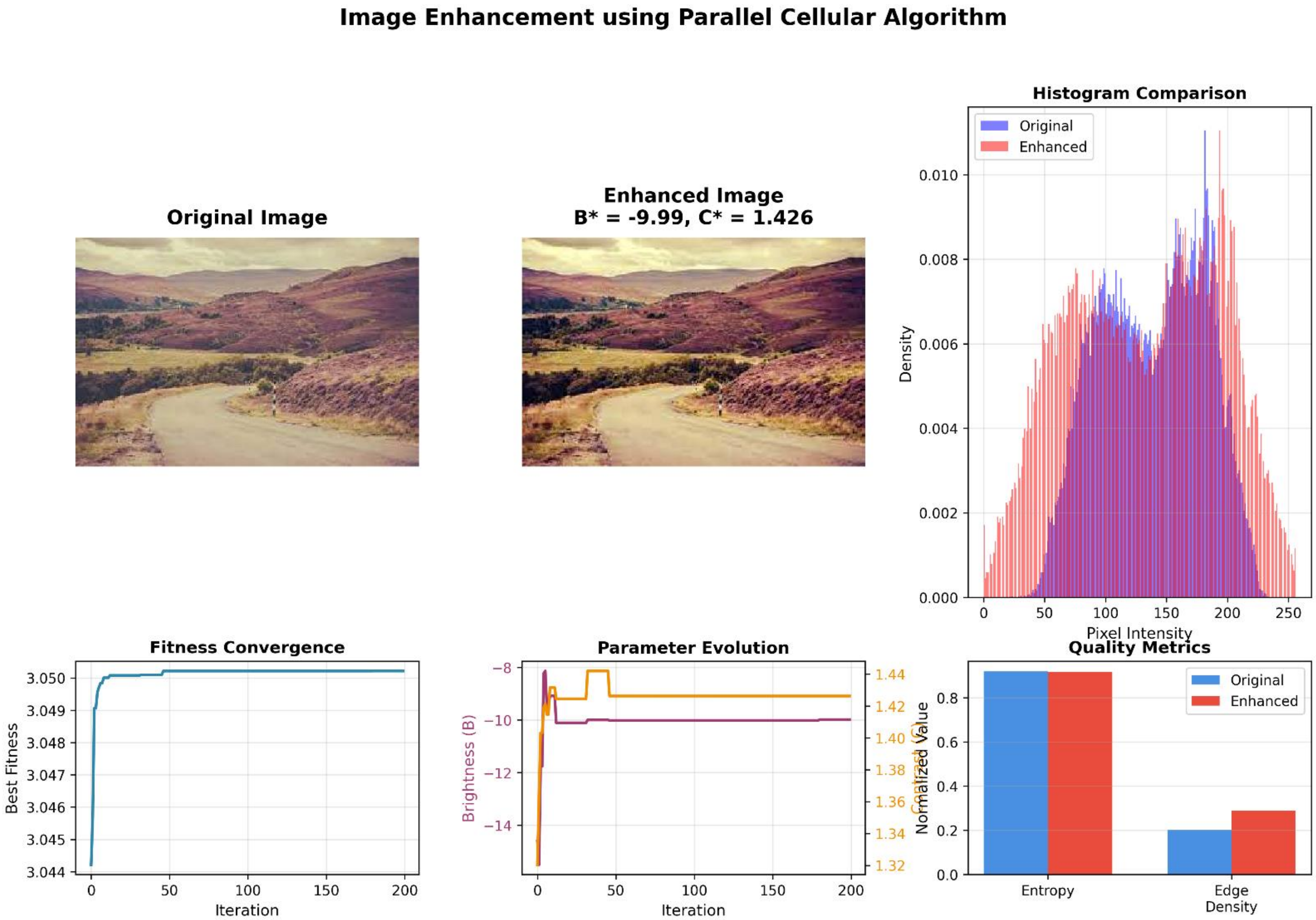


Image Enhancement using Parallel Cellular Algorithm
Original Image
Enhanced Image
B* = -9.99, C* = 1.426
Histogram Comparison
Original
Enhanced
Density
Pixel Intensity
Fitness Convergence
Best Fitness
Iteration
Parameter Evolution
Brightness (B)
Contrast (C)
Iteration
Quality Metrics
Original
Enhanced
Normalized Value
Entropy
Edge Density

Program 7:

Optimization via Gene Expression Algorithms:

Bafna Gold
Date: Page:

Gene Expression Algorithm

Input

Objective function $f(x)$

Population size (N)

Number of gene per sequence (G)

Mutation rate (P_m)

Cross Over rate (P_c)

Maximum generation (MaxGen)

Output

Best Solution and its fitness

1. Initialize a population P of N random genetic sequence

2. for each sequence s in P

 Express s into solution x

 Evaluate fitness $f(x)$

3. Set bestsol = best sequen in P

4. for Generation = 1 to MaxGen do:

 a. Selection

 - Select parent sequence based on fitness

 b. Crossover

 with probability pc, recombine pairs of parent to create offspring sequence

 c. Mutation

 with probability Pm, randomly alter genes in offspring

d. Gene Expression:
Translate offspring sequence into solution

e. Evaluate Fitness
Compute $f(x)$ for each offspring

f. Replacement
Form new pop from offspring

g. Update Best
If better solution found, update best solution

End for

5. Return best solution & its fitness

```
import random
import math
import matplotlib.pyplot as plt

# --- Problem Setup ---
# Coordinates of delivery houses (x, y)
houses = [
    (0, 0),   # Depot (start and end point)
    (2, 3),
    (5, 4),
    (1, 6),
    (7, 2),
    (6, 6)
]

NUM_HOUSES = len(houses)
POP_SIZE = 50
GENERATIONS = 200
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
```

```
# --- Distance Calculation ---
def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def route_distance(route):
    total = 0
    for i in range(len(route)-1):
        total += distance(houses[route[i]], houses[route[i+1]])
    return total

# --- Fitness Function ---
def fitness(route):
    return 1 / (route_distance(route) + 1e-6)

# --- Initialize Population ---
def create_route():
    route = list(range(1, NUM_HOUSES))  # houses except depot
    random.shuffle(route)
    return [0] + route + [0]  # start and end at depot

def init_population():
    return [create_route() for _ in range(POP_SIZE)]

# --- Selection (Tournament) ---
def selection(population):
    tournament = random.sample(population, 5)
    tournament.sort(key=lambda r: fitness(r), reverse=True)
    return tournament[0]

# --- Crossover (Order Crossover - OX) ---
def crossover(parent1, parent2):
    if random.random() > CROSSOVER_RATE:
        return parent1[:]

    start, end = sorted(random.sample(range(1, NUM_HOUSES), 2))
    child = [None] * len(parent1)

    # Copy segment from parent1
    child[start:end] = parent1[start:end]

    # Fill remaining positions from parent2 in order
    fill_values = [g for g in parent2 if g not in child]
    fill_positions = [i for i in range(1, NUM_HOUSES) if child[i] is None]

    for pos, val in zip(fill_positions, fill_values):
        child[pos] = val
```

```
    # Ensure start and end are depot
    child[0] = 0
    child[-1] = 0
    return child

# --- Mutation (Swap) ---
def mutate(route):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(1, NUM_HOUSES), 2)
        route[i], route[j] = route[j], route[i]
    # Ensure depot at start and end
    route[0] = 0
    route[-1] = 0
    return route

# --- GEA Algorithm ---
def gene_expression_algorithm():
    population = init_population()
    best_route = min(population, key=lambda r: route_distance(r))
    best_distance = route_distance(best_route)

    for gen in range(GENERATIONS):
        new_population = []
        for _ in range(POP_SIZE):
            parent1 = selection(population)
            parent2 = selection(population)
            child = crossover(parent1, parent2)
            child = mutate(child)
            # Ensure all genes are integers (no None)
            if None in child:
                child = create_route()
            new_population.append(child)

        population = new_population
        current_best = min(population, key=lambda r: route_distance(r))
        current_dist = route_distance(current_best)

        if current_dist < best_distance:
            best_distance = current_dist
            best_route = current_best

        if gen % 20 == 0:
            print(f"Generation {gen} | Best Distance: {best_distance:.2f}")

    return best_route, best_distance
```

```
# --- Run the Algorithm ---
best_route, best_distance = gene_expression_algorithm()
print("\nBest Route Found:", best_route)
print("Best Distance:", round(best_distance, 2))

# --- Plot the Best Route ---
x = [houses[i][0] for i in best_route]
y = [houses[i][1] for i in best_route]

plt.figure(figsize=(6,6))
plt.plot(x, y, marker='o', color='blue')
for idx, (xi, yi) in enumerate(houses):
    plt.text(xi+0.1, yi+0.1, f"{idx}", fontsize=12)
plt.title("Optimized Delivery Route")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter

```
[Running] python -u "c:\Users\BMSCE\Documents\1BM23CS159\BIS\LAB2(2)\GEA(Route Optimisation).py"
Generation 0 | Best Distance: 20.41
Generation 20 | Best Distance: 7.21
Generation 40 | Best Distance: 7.21
Generation 60 | Best Distance: 7.21
Generation 80 | Best Distance: 7.21
Generation 100 | Best Distance: 7.21
Generation 120 | Best Distance: 7.21
Generation 140 | Best Distance: 7.21
Generation 160 | Best Distance: 7.21
Generation 180 | Best Distance: 7.21

Best Route Found: [0, 0, 0, 0, 1, 0, 0]
Best Distance: 7.21

[Done] exited with code=0 in 18.211 seconds
```

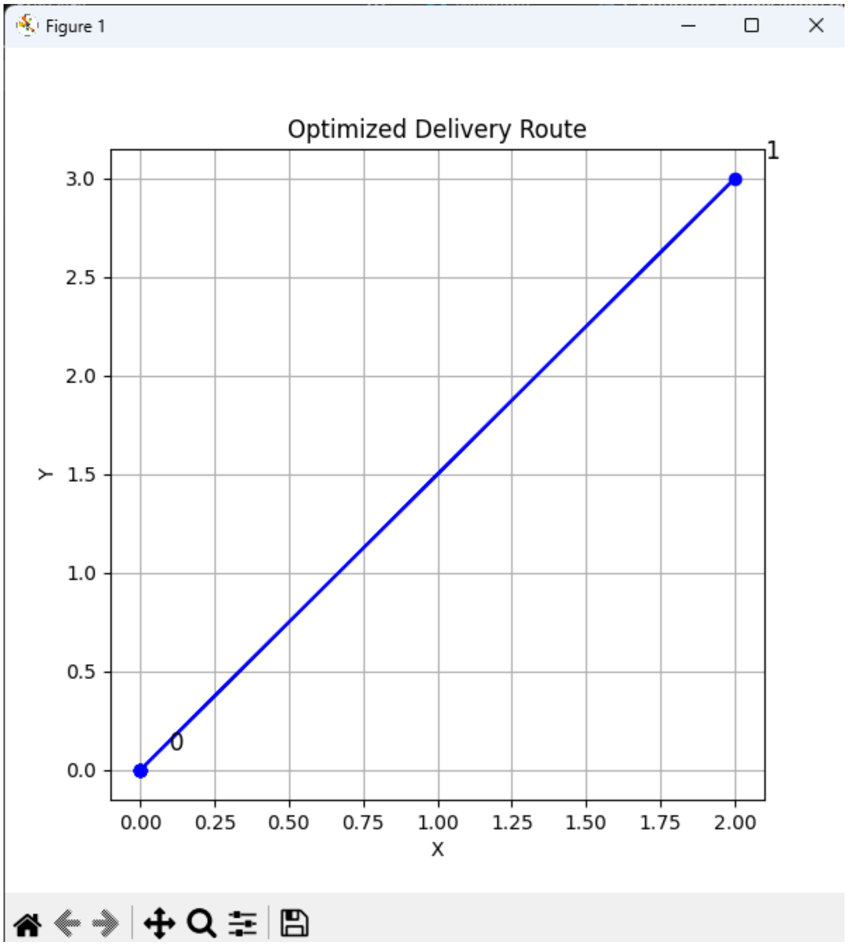


Figure 1
Optimized Delivery Route
1
0
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
X