VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PRASOBH RATNA SHAKYA(1BM23CS243)

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
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 **PRASOBH RATNA SHAKYA (1BM23CS243),** 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.

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

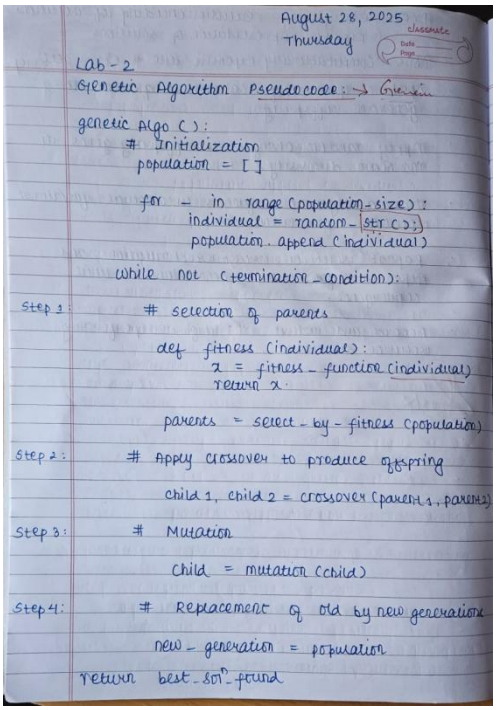
Index

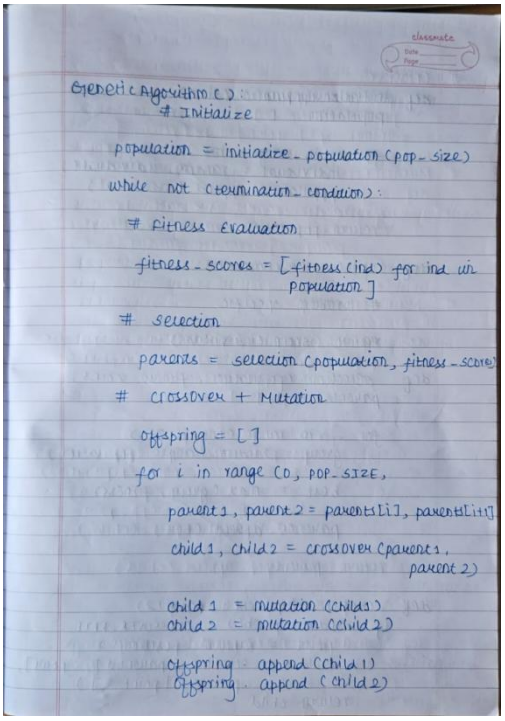
Github Link:
https://github.com/PrasobhRatnaShakya/BIS-Lab

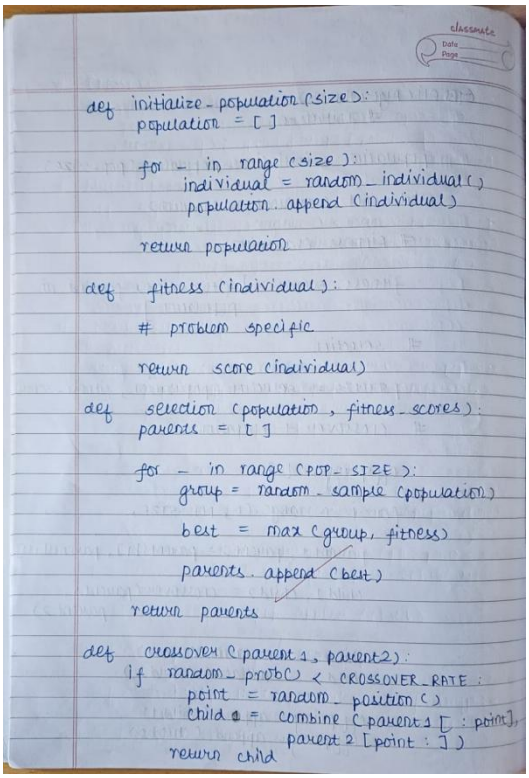
Program 1

Problem statement: Use the Genetic Algorithm for Traffic Management.

Algorithm:







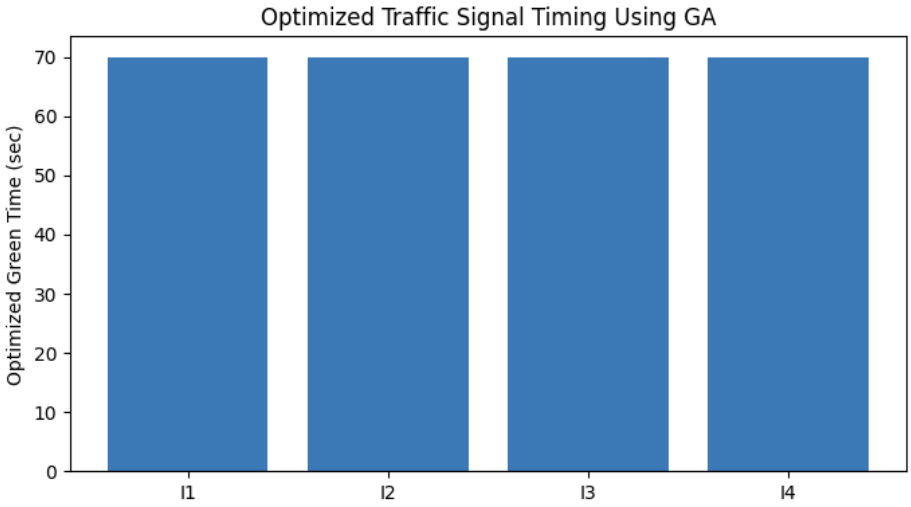
Code:

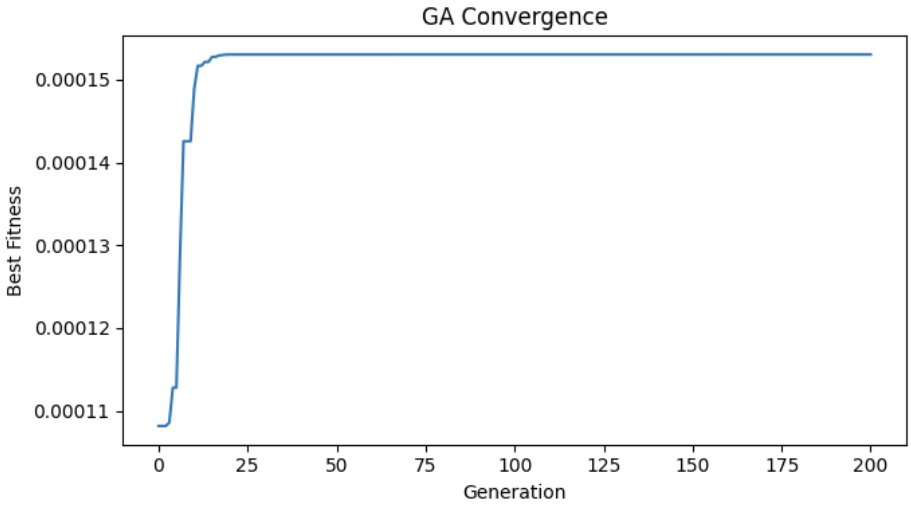
```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(42)

def simulate_traffic(green_times, flow_rates):
    total_wait = 0.0
    for i in range(len(flow_rates)):
        arrival = flow_rates[i]
        green = green_times[i]
        cycle = 90
        red = cycle - green
        wait = arrival * (red / cycle)**2 * cycle
        total_wait += wait
    return float(total_wait)

def fitness(ind, flow_rates):
    w = simulate_traffic(ind, flow_rates)
    return 1.0 / (1.0 + w)

def run_ga(flow_rates, bounds, cfg):
    pop_size, generations, elite_frac, tournament_k, mutation_rate,
mutation_scale, crossover_rate, rng = cfg

    def clip(ind):
        for i,(lo,hi) in enumerate(bounds):
            ind[i] = np.clip(ind[i], lo, hi)
        return ind

    def random_ind():
        return np.array([rng.uniform(lo,hi) for lo,hi in bounds],
float)

    def tournament(pop, fit, k):
        idx = rng.integers(0, len(pop), k)
        b = idx[np.argmax(fit[idx])]
        return pop[b].copy()

    def crossover(p1, p2):
        a = rng.uniform(0,1,p1.shape)
        c1 = a*p1 + (1-a)*p2
        c2 = a*p2 + (1-a)*p1
        return c1, c2

    def mutate(ind):
        ind = ind.copy()
```

```
        for i,(lo,hi) in enumerate(bounds):
            if rng.random() < mutation_rate:
                span = hi-lo
                ind[i] += rng.normal(0, mutation_scale*span)
        return clip(ind)

    pop = np.array([random_ind() for _ in range(pop_size)])
    fit = np.array([fitness(ind, flow_rates) for ind in pop])
    best_curve = [float(np.max(fit))]
    n_elite = max(1, int(pop_size * elite_frac))

    for _ in range(generations):
        elite_idx = np.argsort(fit)[-n_elite:]
        elites = pop[elite_idx].copy()
        next_pop = []
        while len(next_pop) < pop_size - n_elite:
            p1 = tournament(pop, fit, tournament_k)
            p2 = tournament(pop, fit, tournament_k)
            if rng.random() < crossover_rate:
                c1, c2 = crossover(p1,p2)
            else:
                c1, c2 = p1.copy(), p2.copy()
            next_pop.append(mutate(c1))
            if len(next_pop) < pop_size - n_elite:
                next_pop.append(mutate(c2))
        pop = np.vstack([elites, np.array(next_pop)])
        fit = np.array([fitness(ind, flow_rates) for ind in pop])
        best_curve.append(float(np.max(fit)))

    best_idx = int(np.argmax(fit))
    best = pop[best_idx].copy()
    return best, best_curve

if __name__ == "__main__":
    intersections = 4
    flow_rates = np.array([300, 520, 400, 250], float)
    bounds = [(10,70)] * intersections
    cfg = (200, 200, 0.07, 3, 0.18, 0.12, 0.9,
np.random.default_rng(2025))

    best, curve = run_ga(flow_rates, bounds, cfg)
    x = np.arange(len(best))

    plt.figure(figsize=(7,4))
    plt.bar(x, best)
    plt.xticks(x, [f"I{i+1}" for i in x])
    plt.ylabel("Optimized Green Time (sec)")
    plt.title("Optimized Traffic Signal Timing Using GA")
```

```
plt.tight_layout()
plt.show()

plt.figure(figsize=(7,4))
plt.plot(curve)
plt.xlabel("Generation")
plt.ylabel("Best Fitness")
plt.title("GA Convergence")
plt.tight_layout()
plt.show()
```

Output:


Optimized Traffic Signal Timing Using GA
Optimized Green Time (sec)
0
10
20
30
40
50
60
70
I1
I2
I3
I4

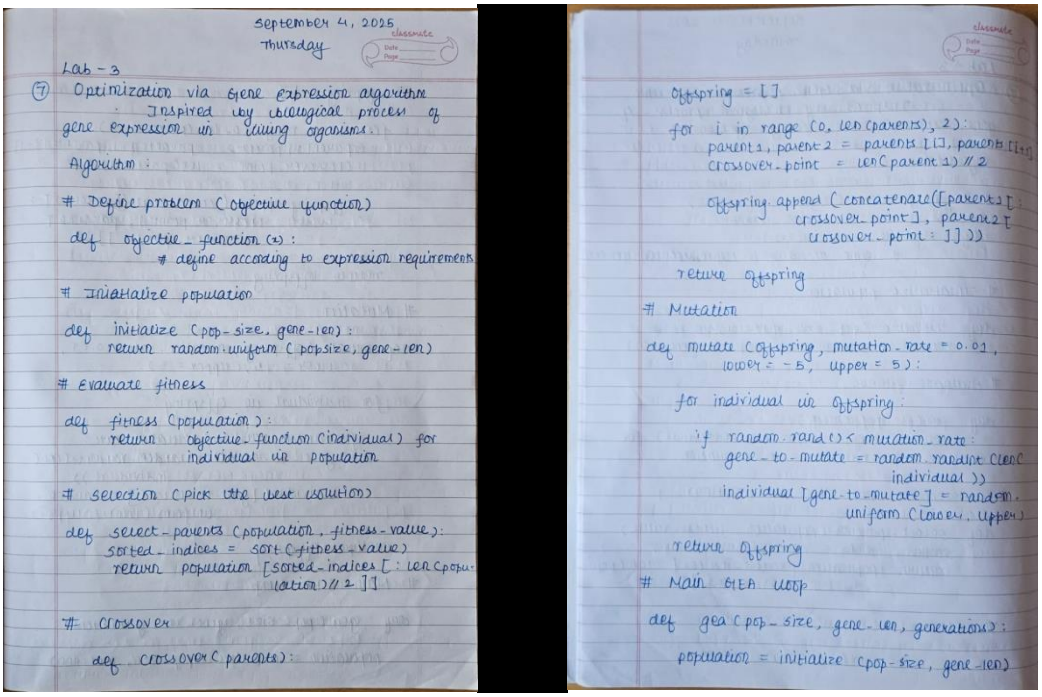

GA Convergence
Best Fitness
0.00011
0.00012
0.00013
0.00014
0.00015
0
25
50
75
100
125
150
175
200
Generation

Program 2

Problem statement Use the Gene Expression Algorithm for specific Genetic Algorithm.

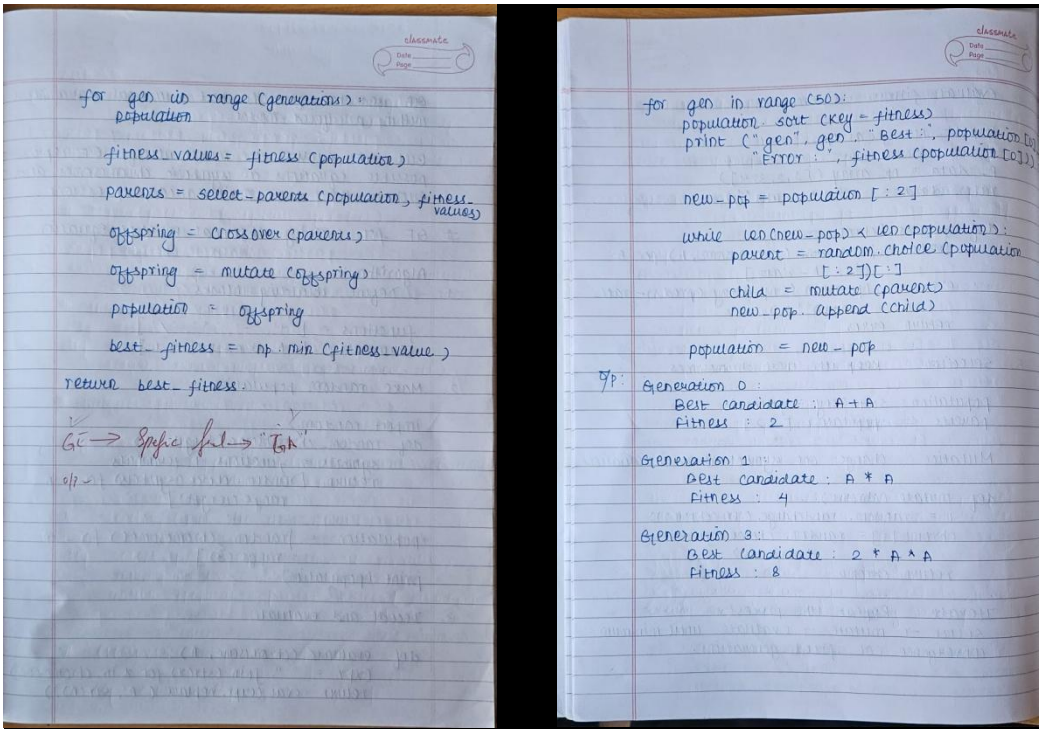
GE for evolving Chemical Kinetics Equation:

Algorithm:



```
September 4, 2025
Thursday
Lab - 3
(7) Optimization via Gene expression algorithm
    Inspired by biological process of
gene expression in living organisms.
Algorithm:

# Define problem (objective function)

def objective_function(x):
    # define according to expression requirements

# Initialize population

def initialize(pop_size, gene_len):
    return random.uniform(popsize, gene_len)

# Evaluate fitness

def fitness(population):
    return objective_function(individual) for
        individual in population

# selection (pick the best solution)

def select_parents(population, fitness_value):
    sorted_indices = sort(fitness_value)
    return population[sorted_indices[: len(popu-
                                    lation)//2]]

# Crossover

def crossover(parents):
```

```
    offspring = []

    for i in range(0, len(parents), 2):
        parent1, parent2 = parents[i], parents[i+1]
        crossover_point = len(parent1) // 2

        offspring.append(concatenate([parent1[:
                crossover_point], parent2[
                crossover_point:]]))

    return offspring

# Mutation

def mutate(offspring, mutation_rate = 0.01,
           lower = -5, upper = 5):

    for individual in offspring:

        if random.rand() < mutation_rate:
            gene_to_mutate = random.randint(len(
                                individual))
            individual[gene_to_mutate] = random.
                uniform(lower, upper)

    return offspring

# Main GEA loop

def gea(pop_size, gene_len, generations):

    population = initialize(pop_size, gene_len)
```



```
    for gen in range(generations):
        population

        fitness_values = fitness(population)

        parents = select_parents(population, fitness_
                                    values)

        offspring = crossover(parents)

        offspring = mutate(offspring)

        population = offspring

        best_fitness = np.min(fitness_value)

    return best_fitness.

GE -> Specific fn -> "GEA"

o/p -
```

```
    for gen in range(50):
        population.sort(key = fitness)
        print("gen", gen, "Best:", population[0],
              "Error:", fitness(population[0]))

        new_pop = population[:2]

        while len(new_pop) < len(population):
            parent = random.choice(population
                                   [:2])[:]
            child = mutate(parent)
            new_pop.append(child)

        population = new_pop

o/p:
Generation 0:
    Best candidate: A + A
    Fitness: 2

Generation 1:
    Best candidate: A * A
    Fitness: 4

Generation 3:
    Best candidate: 2 * A * A
    Fitness: 8
```

Code:

```
# Evolving a simple chemical kinetics rate law using a toy GEP/GP-
style symbolic regression
# Requirements: numpy, pandas, matplotlib

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import math

# ----------------------------
# 1. Generate synthetic kinetics data (A -> B, first-order: dA/dt =
-k*A)
# ----------------------------
def true_rhs(A, k=0.5):
    return -k * A

def simulate_kinetics(A0=1.0, k=0.5, t_end=10.0, dt=0.1):
    t = np.arange(0, t_end + dt, dt)
    A = np.zeros_like(t)
    A[0] = A0
    for i in range(1, len(t)):
        # simple explicit Euler integrator
        A[i] = A[i-1] + true_rhs(A[i-1], k) * dt
    return t, A

t, A = simulate_kinetics()

# build training data: input = A(t), target = -dA/dt (i.e. rate)
dA_dt = np.gradient(A, t)
target_rate = -dA_dt

data = pd.DataFrame({"t": t, "A": A, "rate": target_rate})

# ----------------------------
# 2. Define a simple "gene expression" representation:
#    expression trees built from a small function set
# ----------------------------
FUNCTIONS = [
    ("add", 2, lambda x, y: x + y),
    ("sub", 2, lambda x, y: x - y),
    ("mul", 2, lambda x, y: x * y),
    ("div", 2, lambda x, y: x / (y + 1e-8)),  # protected division
    ("sin", 1, lambda x: np.sin(x)),
    ("exp", 1, lambda x: np.exp(np.clip(x, -5, 5))),  # clip to
avoid overflow
```

```
]

TERMINALS = ["A", "const"]  # concentration A and an ephemeral
constant

MAX_DEPTH = 4

def random_const():
    return random.uniform(-2.0, 2.0)

class Node:
    def __init__(self, kind, name=None, arity=0, func=None,
value=None, children=None):
        self.kind = kind      # "func" or "term"
        self.name = name
        self.arity = arity
        self.func = func
        self.value = value    # for const
        self.children = children or []

    def copy(self):
        return Node(
            self.kind,
            self.name,
            self.arity,
            self.func,
            self.value,
            [c.copy() for c in self.children],
        )

def random_tree(depth=0):
    if depth >= MAX_DEPTH or (depth > 0 and random.random() < 0.3):
        term = random.choice(TERMINALS)
        if term == "const":
            return Node("term", "const", value=random_const())
        else:
            return Node("term", "A")
    # function node
    name, arity, func = random.choice(FUNCTIONS)
    children = [random_tree(depth + 1) for _ in range(arity)]
    return Node("func", name, arity, func, children=children)

# -----------------------------
# 3. Evaluation: map A -> rate_hat = f(A)
# -----------------------------
def eval_tree(node, A_values):
    if node.kind == "term":
        if node.name == "A":
```

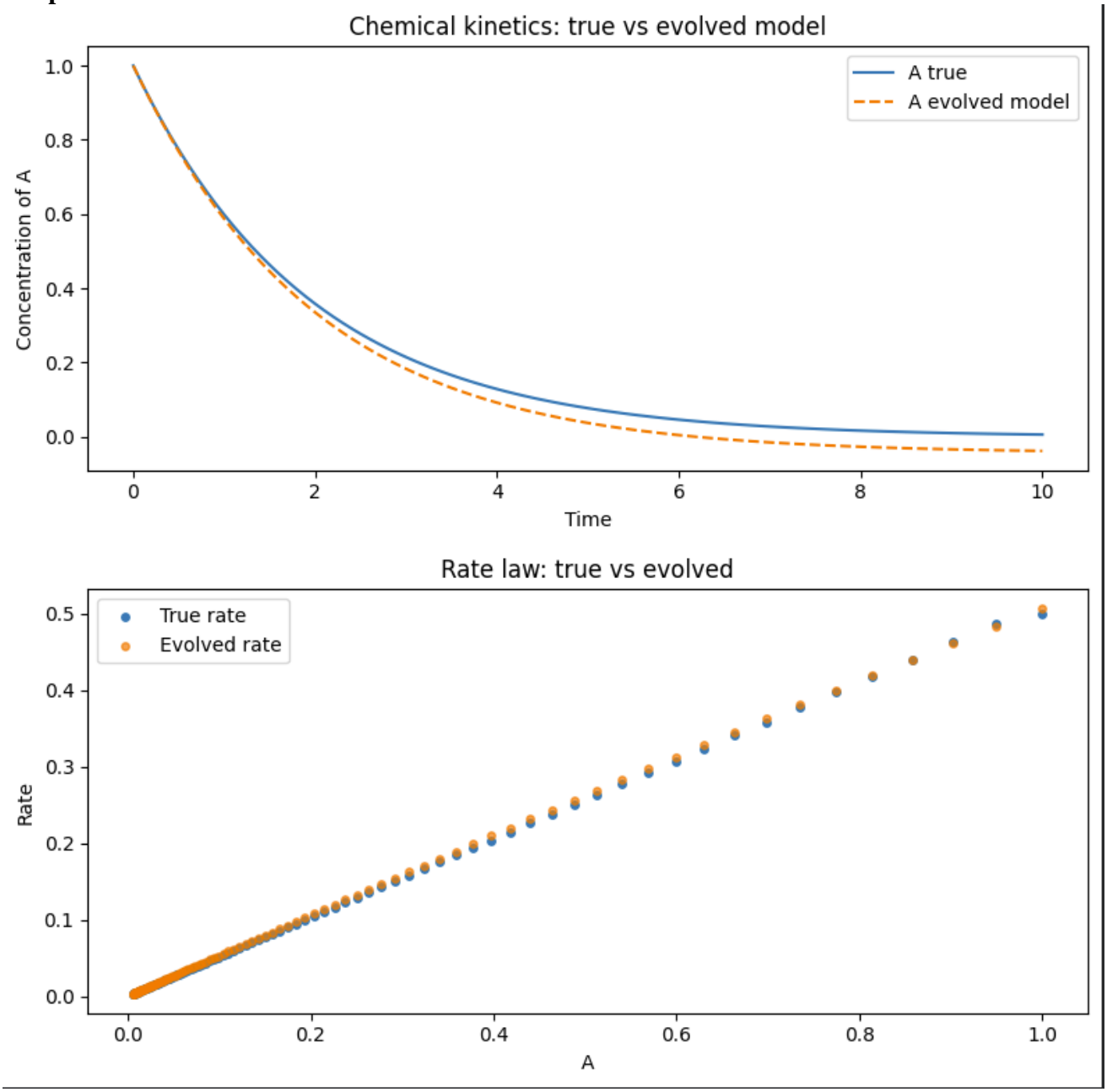
```
            return A_values
        elif node.name == "const":
            return np.full_like(A_values, node.value, dtype=float)
    else:
        if node.arity == 1:
            x = eval_tree(node.children[0], A_values)
            return node.func(x)
        elif node.arity == 2:
            x = eval_tree(node.children[0], A_values)
            y = eval_tree(node.children[1], A_values)
            return node.func(x, y)

def fitness(indiv, A_values, target):
    y_pred = eval_tree(indiv, A_values)
    # mean squared error; penalize NaNs/Infs
    if np.any(np.isnan(y_pred)) or np.any(np.isinf(y_pred)):
        return 1e9
    return float(np.mean((y_pred - target) ** 2))

# ----------------------------
# 4. Genetic operators
# ----------------------------
def random_subtree(node):
    # collect all nodes
    nodes = []
    def collect(n):
        nodes.append(n)
        for c in n.children:
            collect(c)
    collect(node)
    return random.choice(nodes)

def mutate(indiv, pmut=0.2):
    child = indiv.copy()
    if random.random() < pmut:
        # replace a random subtree
        subtree_parent = child
        subtree = random_subtree(child)
        # simple: just replace whole tree
        # (for a minimal example; can refine to track parents)
        return random_tree()
    # small numeric mutation on constants
    def mutate_const(n):
        if n.kind == "term" and n.name == "const":
            if random.random() < 0.5:
                n.value += random.uniform(-0.5, 0.5)
        for c in n.children:
            mutate_const(c)
```

```
    mutate_const(child)
    return child

def crossover(parent1, parent2, pcross=0.7):
    if random.random() > pcross:
        return parent1.copy(), parent2.copy()

    # deep copies
    c1 = parent1.copy()
    c2 = parent2.copy()

    # naive subtree swap: treat root as subtree for this minimal
example
    return c2, c1

# ----------------------------
# 5. Main evolutionary loop
# ----------------------------
POP_SIZE = 50
N_GEN = 40
TOUR_SIZE = 3
MUT_PROB = 0.7
CROSS_PROB = 0.7

# initialize population
population = [random_tree() for _ in range(POP_SIZE)]

def tournament_select(pop, fits, k=TOUR_SIZE):
    idxs = random.sample(range(len(pop)), k)
    best_i = min(idxs, key=lambda i: fits[i])
    return pop[best_i]

A_values = data["A"].values
target = data["rate"].values

best_indiv = None
best_fit = 1e9

for gen in range(N_GEN):
    fits = [fitness(ind, A_values, target) for ind in population]
    # track best
    for ind, f in zip(population, fits):
        if f < best_fit:
            best_fit = f
            best_indiv = ind
    print(f"Gen {gen:03d} best MSE = {best_fit:.6e}")

    # create new population
```

```
    new_pop = []
    while len(new_pop) < POP_SIZE:
        p1 = tournament_select(population, fits)
        p2 = tournament_select(population, fits)
        c1, c2 = crossover(p1, p2, CROSS_PROB)
        c1 = mutate(c1, MUT_PROB)
        c2 = mutate(c2, MUT_PROB)
        new_pop.extend([c1, c2])
    population = new_pop[:POP_SIZE]

# ------------------------------
# 6. Visualize learned rate law & integrated dynamics
# ------------------------------
# Evaluate predicted rate vs true target
rate_hat = eval_tree(best_indiv, A_values)

# integrate dA/dt = -rate_hat(A)
A_model = np.zeros_like(A_values)
A_model[0] = A_values[0]
dt = t[1] - t[0]
for i in range(1, len(A_model)):
    A_model[i] = A_model[i-1] - rate_hat[i-1] * dt

result_df = pd.DataFrame(
    {"t": t, "A_true": A_values, "A_model": A_model,
     "rate_true": target, "rate_hat": rate_hat}
)

# Plot concentration profiles
plt.figure(figsize=(8, 4))
plt.plot(result_df["t"], result_df["A_true"], label="A true")
plt.plot(result_df["t"], result_df["A_model"], "--", label="A
evolved model")
plt.xlabel("Time")
plt.ylabel("Concentration of A")
plt.legend()
plt.title("Chemical kinetics: true vs evolved model")
plt.tight_layout()
plt.show()

# Plot rates
plt.figure(figsize=(8, 4))
plt.scatter(result_df["A_true"], result_df["rate_true"], s=15,
label="True rate")
plt.scatter(result_df["A_true"], result_df["rate_hat"], s=15,
label="Evolved rate", alpha=0.7)
plt.xlabel("A")
plt.ylabel("Rate")
```

```
plt.legend()
plt.title("Rate law: true vs evolved")
plt.tight_layout()
plt.show()
```

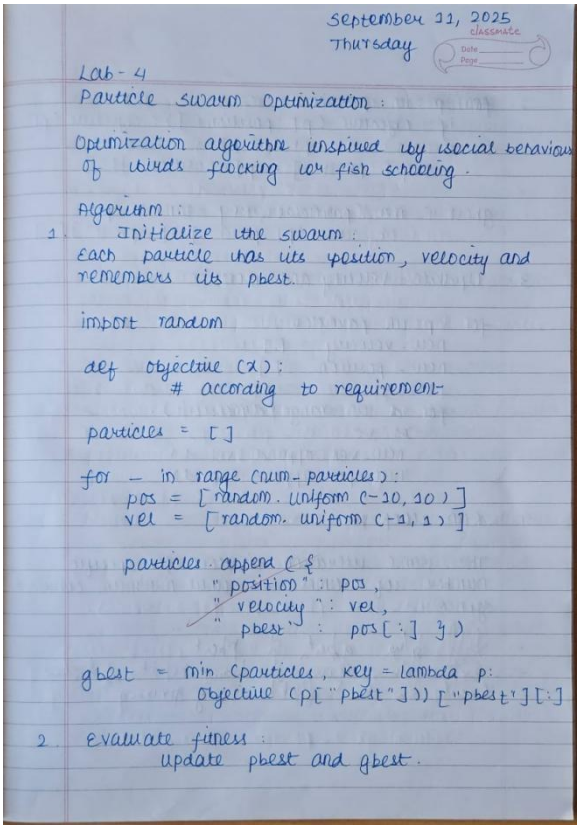
Output:



Program 3

Problem statement: Use Particle Swarm Optimization Algorithm in the field of Image Processing.

Algorithm:



September 11, 2025
Thursday

Lab - 4
Particle Swarm Optimization :

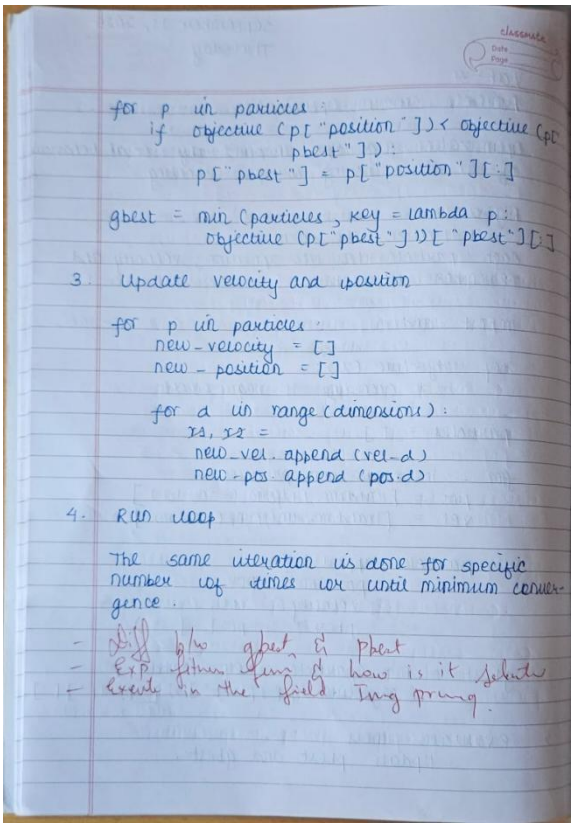
Optimization algorithm inspired by social behaviour of birds flocking or fish schooling.

Algorithm :

1. Initialize the swarm :
Each particle has its position, velocity and remembers its pbest.

```
import random

def objective(x):
    # according to requirement

particles = []

for _ in range(num_particles):
    pos = [random.uniform(-10, 10)]
    vel = [random.uniform(-1, 1)]

    particles.append({
        "position": pos,
        "velocity": vel,
        "pbest": pos[:]})

gbest = min(particles, key=lambda p:
        objective(p["pbest"]))["pbest"][:]
```

2. evaluate fitness :
update pbest and gbest.



```
for p in particles:
    if objective(p["position"]) < objective(p[
                "pbest"]):
        p["pbest"] = p["position"][:]

gbest = min(particles, key=lambda p:
        objective(p["pbest"]))["pbest"][:]
```

3. Update velocity and position

```
for p in particles:
    new_velocity = []
    new_position = []

    for d in range(dimensions):
        r1, r2 =
        new_vel.append(vel_d)
        new_pos.append(pos_d)
```

4. Run loop

The same iteration is done for specific number of times or until minimum convergence.

- Diff b/w gbest & Pbest
- Exp fitness func & how is it selected
- Events in the field Img proc.

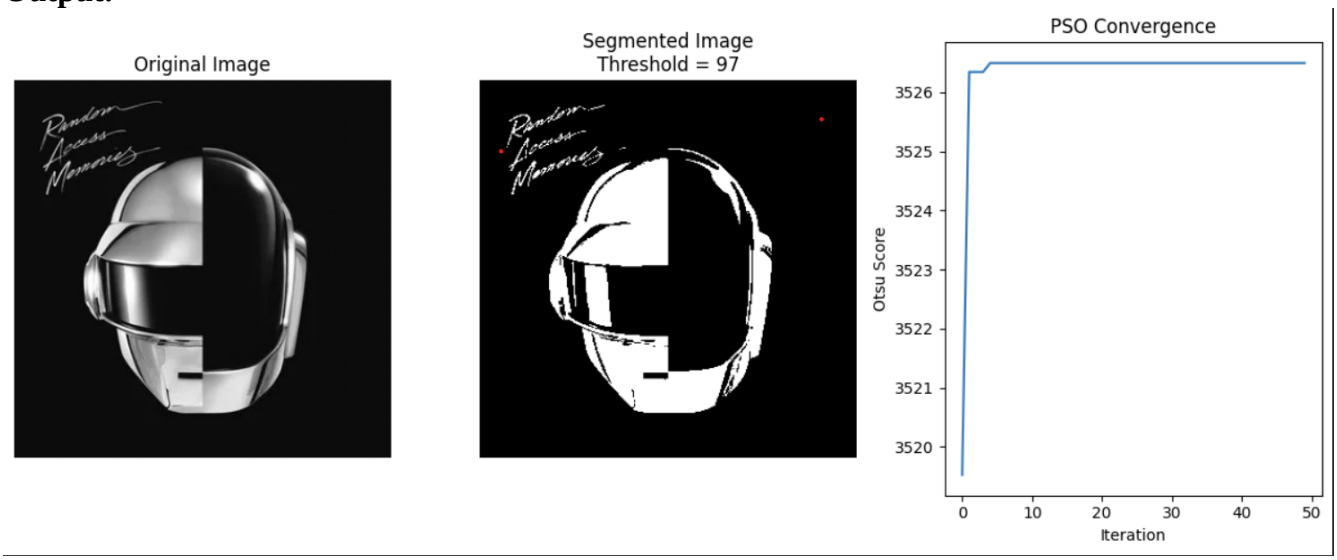
Code:

```
# Step 1: Upload the image manually
from google.colab import files
uploaded = files.upload()  # Upload dialog will pop up

# Get uploaded filename
img_path = list(uploaded.keys())[0]
print(f"Uploaded image: {img_path}")

# Step 2: Import packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
import random

# Step 3: Load image in grayscale
img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
if img is None:
    raise FileNotFoundError("Image not found or could not be
read.")
pixels = img.flatten()

# Step 4: Define fitness function (Otsu between-class variance)
def otsu_between_class_variance(threshold, pixels):
    threshold = int(np.clip(threshold, 0, 255))
    hist, _ = np.histogram(pixels, bins=256, range=(0,256))
    total = pixels.size

    w0 = hist[:threshold].sum() / total
    w1 = hist[threshold:].sum() / total

    if w0 == 0 or w1 == 0:
        return 0.0

    mu0 = np.sum(np.arange(0, threshold) * hist[:threshold]) /
(hist[:threshold].sum() + 1e-8)
    mu1 = np.sum(np.arange(threshold, 256) * hist[threshold:]) /
(hist[threshold:].sum() + 1e-8)

    return w0 * w1 * (mu0 - mu1) ** 2

def fitness(threshold, pixels):
    return -otsu_between_class_variance(threshold, pixels)

# Step 5: PSO implementation
class Particle:
    def __init__(self, min_x, max_x):
```

```
        self.position = random.uniform(min_x, max_x)
        self.velocity = random.uniform(-10, 10)
        self.best_position = self.position
        self.best_value = float("inf")

def pso_optimize(
    pixels,
    n_particles=20,
    n_iters=50,
    min_x=0,
    max_x=255,
    w=0.7,
    c1=1.5,
    c2=1.5,
):
    swarm = [Particle(min_x, max_x) for _ in range(n_particles)]
    g_best_position = swarm[0].position
    g_best_value = float("inf")

    history = []

    for it in range(n_iters):
        for p in swarm:
            f = fitness(p.position, pixels)
            if f < p.best_value:
                p.best_value = f
                p.best_position = p.position
            if f < g_best_value:
                g_best_value = f
                g_best_position = p.position

        for p in swarm:
            r1 = random.random()
            r2 = random.random()
            cognitive = c1 * r1 * (p.best_position - p.position)
            social = c2 * r2 * (g_best_position - p.position)
            p.velocity = w * p.velocity + cognitive + social
            p.position += p.velocity
            p.position = max(min_x, min(max_x, p.position))

        history.append((it, g_best_position, g_best_value))
        print(f"Iteration {it}: Best threshold =
{g_best_position:.2f}, Fitness = {g_best_value:.6e}")

    hist_df = pd.DataFrame(history, columns=["Iteration", "Best
Threshold", "Fitness"])
    return int(round(g_best_position)), hist_df
```

```
# Step 6: Run PSO to find optimal threshold
best_threshold, history_df = pso_optimize(pixels)

print(f"Optimal threshold found by PSO: {best_threshold}")

# Step 7: Apply threshold and visualize results
_, segmented = cv2.threshold(img, best_threshold, 255,
cv2.THRESH_BINARY)

plt.figure(figsize=(12, 5))
plt.subplot(1, 3, 1)
plt.imshow(img, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(segmented, cmap='gray')
plt.title(f'Segmented Image\nThreshold = {best_threshold}')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.plot(history_df["Iteration"], -history_df["Fitness"])
plt.xlabel("Iteration")
plt.ylabel("Otsu Score")
plt.title("PSO Convergence")

plt.tight_layout()
plt.show()
```

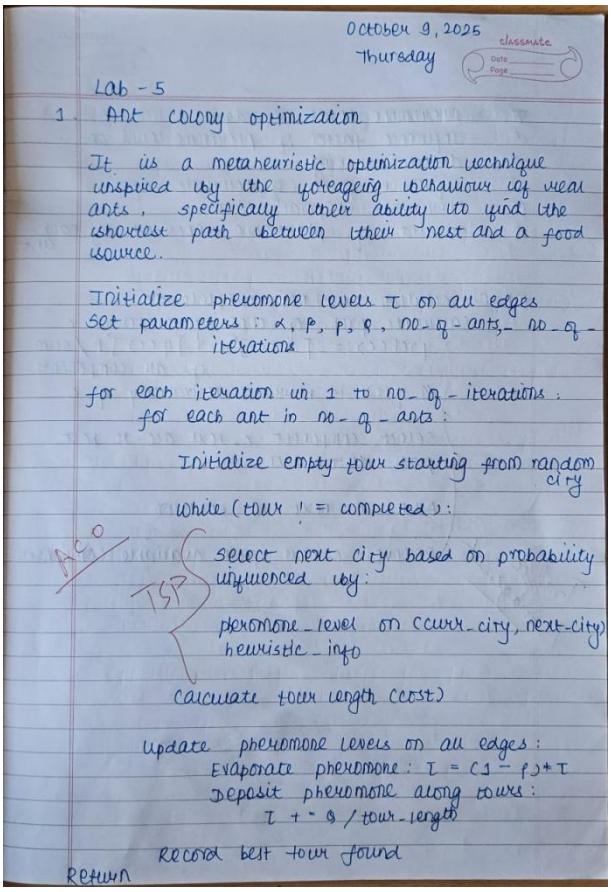
Output:


Original Image
Segmented Image
Threshold = 97
PSO Convergence
Otsu Score
3526
3525
3524
3523
3522
3521
3520
0
10
20
30
40
50
Iteration

Program 4

Problem statement: Use Ant Colony Optimization Algorithm to optimize traffic management.

Algorithm:



October 9, 2025
Thursday

Lab - 5

1. Ant colony optimization

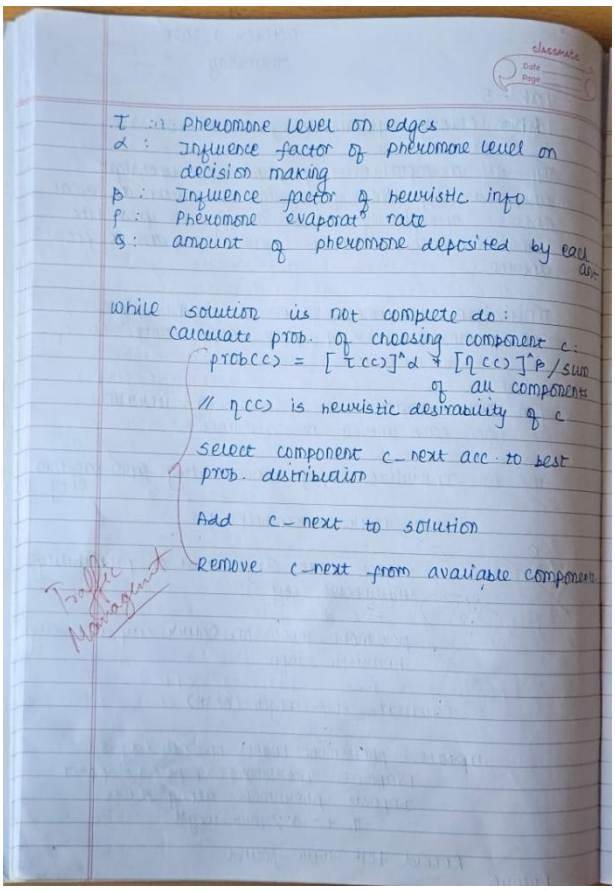
It is a metaheuristic optimization technique inspired by the foraging behaviour of real ants, specifically their ability to find the shortest path between their nest and a food source.

Initialize pheromone levels τ on all edges
Set parameters : α, β, ρ, Q, no_of_ants, no_of_iterations

for each iteration in 1 to no_of_iterations :
  for each ant in no_of_ants :
    Initialize empty tour starting from random city
    while (tour != completed) :
      select next city based on probability influenced by:
      pheromone_level on (curr_city, next_city)
      heuristic_info
    Calculate tour length (cost)

  Update pheromone levels on all edges :
    Evaporate pheromone : $\tau = (1 - \rho) * \tau$
    Deposit pheromone along tours :
      τ += Q / tour_length
  Record best tour found
Return

ACO
TSP



τ : Pheromone level on edges
α : Influence factor of pheromone level on decision making
β : Influence factor of heuristic info
ρ : Pheromone evaporat° rate
Q : amount of pheromone deposited by each ant

while solution is not complete do :
  Calculate prob. of choosing component c :
    prob(c) = [τ(c)]^α * [η(c)]^β / sum of all components
    // η(c) is heuristic desirability of c
  Select component c_next acc to best prob. distribution
  Add c_next to solution
  Remove c_next from available components

Traffic Management

Code:

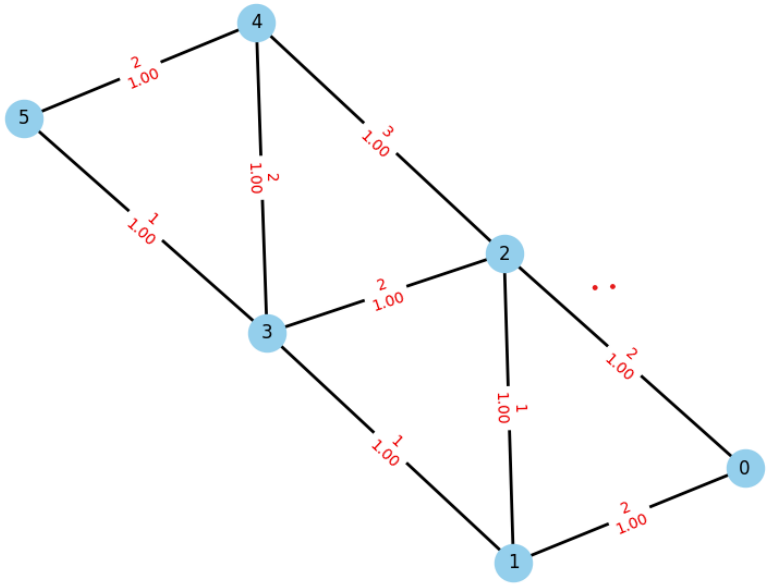
```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd

# Traffic network graph (nodes: intersections, edges: roads)
# We'll use NetworkX to create and visualize the graph
def create_traffic_graph():
    G = nx.Graph()
    # Add nodes (intersections)
    for i in range(6):
        G.add_node(i)
    # Add edges (roads) with initial distance (weight) and
pheromone level
    edges = [
        (0, 1, 2), (0, 2, 2), (1, 2, 1), (1, 3, 1),
        (2, 3, 2), (2, 4, 3), (3, 4, 2), (3, 5, 1),
        (4, 5, 2)
    ]
    for (u, v, dist) in edges:
        G.add_edge(u, v, distance=dist, pheromone=1.0)
    return G

# Visualize graph with pheromone intensity on edges
def visualize_graph(G, title="Traffic Network"):
    pos = nx.spring_layout(G, seed=42)
    edge_labels = {(u, v): f"{d['distance']}\n{d['pheromone']:.2f}"
for u, v, d in G.edges(data=True)}
    pheromones = np.array([d['pheromone'] for _, _, d in
G.edges(data=True)])
    # Normalize pheromones for edge width
    widths = 2 + 5 * (pheromones - pheromones.min()) /
(pheromones.max() - pheromones.min() + 1e-6)

    plt.figure(figsize=(8,6))
    nx.draw(G, pos, with_labels=True, node_size=600,
node_color='skyblue', width=widths)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_color='red')
    plt.title(title)
    plt.show()

# Ant Colony Optimization for path from start to end node
def ant_colony_optimization(G, start, end, n_ants=10,
n_iterations=50, alpha=1, beta=2, rho=0.1, Q=1):
    """
    alpha: pheromone importance
```

```
    beta: heuristic importance (1/distance)
    rho: pheromone evaporation rate
    Q: pheromone deposit factor
    """
    nodes = list(G.nodes())
    best_path = None
    best_length = float('inf')

    for iteration in range(n_iterations):
        all_paths = []
        all_lengths = []

        for ant in range(n_ants):
            path = [start]
            visited = set(path)

            while path[-1] != end:
                current = path[-1]
                neighbors = list(G.neighbors(current))
                # Allowed moves: neighbors not visited yet
                allowed = [n for n in neighbors if n not in
visited]

                if len(allowed) == 0:
                    # Dead end, backtrack randomly (or break)
                    break

                # Calculate probabilities based on pheromone and
heuristic
                pheromones = np.array([G[current][nbr]['pheromone']
for nbr in allowed])
                distances = np.array([G[current][nbr]['distance']
for nbr in allowed])
                heuristic = 1 / distances

                prob = (pheromones ** alpha) * (heuristic ** beta)
                prob /= prob.sum()

                next_node = np.random.choice(allowed, p=prob)
                path.append(next_node)
                visited.add(next_node)

            # Calculate path length
            length = sum(G[path[i]][path[i+1]]['distance'] for i in
range(len(path)-1))
            all_paths.append(path)
            all_lengths.append(length)
```

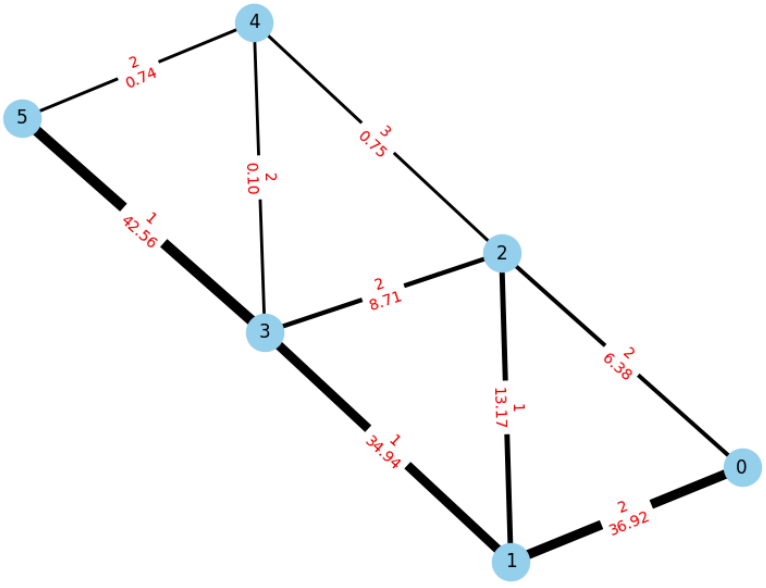
```
            # Update best path
            if length < best_length and path[-1] == end:
                best_length = length
                best_path = path

        # Evaporate pheromones
        for u, v in G.edges():
            G[u][v]['pheromone'] *= (1 - rho)
            if G[u][v]['pheromone'] < 0.1:
                G[u][v]['pheromone'] = 0.1  # minimum pheromone

        # Deposit pheromones based on paths found
        for path, length in zip(all_paths, all_lengths):
            if path[-1] == end:
                deposit = Q / length
                for i in range(len(path)-1):
                    u, v = path[i], path[i+1]
                    G[u][v]['pheromone'] += deposit

        print(f"Iteration {iteration+1}/{n_iterations}, best
length: {best_length:.2f}")

    return best_path, best_length

# Main execution
if __name__ == "__main__":
    G = create_traffic_graph()
    visualize_graph(G, title="Initial Traffic Network (Distance /
Pheromone)")

    start_node = 0
    end_node = 5
    best_path, best_length = ant_colony_optimization(G, start_node,
end_node, n_ants=20, n_iterations=100)

    print(f"\nBest path found: {best_path} with length
{best_length:.2f}")
    visualize_graph(G, title="Traffic Network after ACO
Optimization")
```

Output:

Initial Traffic Network (Distance / Pheromone)



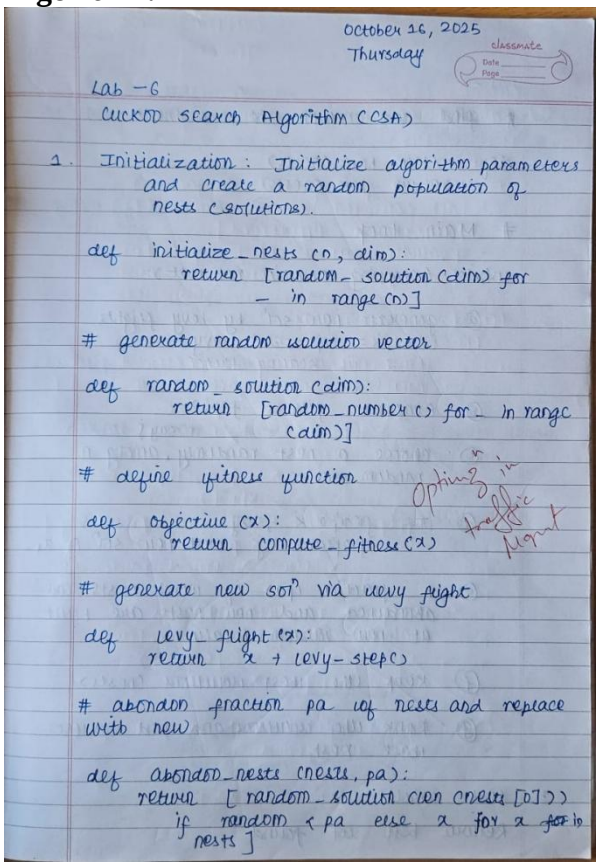
Traffic Network after ACO Optimization



Program 5

Problem statement: Use Cuckoo Search Algorithm to optimize Traffic Management.

Algorithm:



October 16, 2025
Thursday

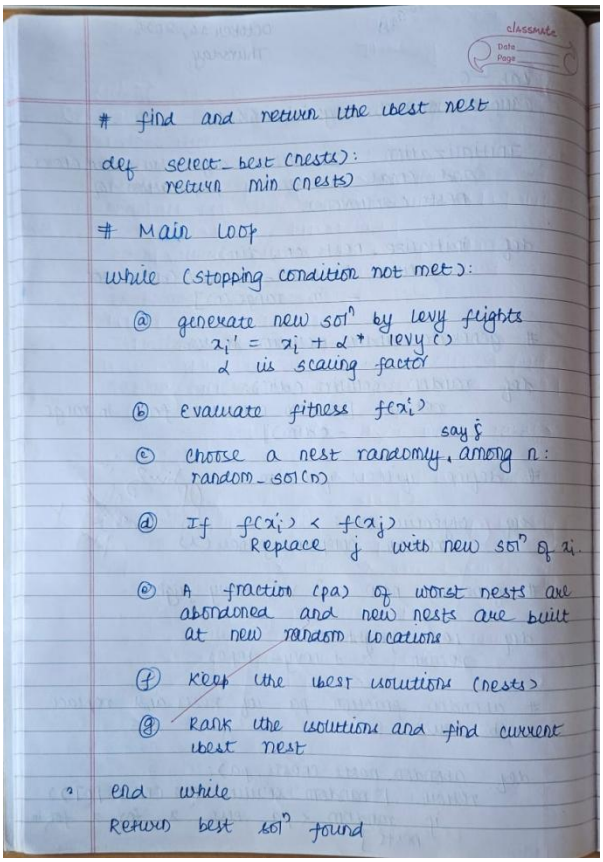
Lab - 6

Cuckoo Search Algorithm (CSA)

1. Initialization: Initialize algorithm parameters and create a random population of nests (solutions).

```
def initialize_nests(n, dim):
    return [random_solution(dim) for _ in range(n)]

# generate random solution vector

def random_solution(dim):
    return [random_number() for _ in range(dim)]

# define fitness function

def objective(x):
    return compute_fitness(x)

# generate new sol^n via levy flight

def levy_flight(x):
    return x + levy_step()

# abandon fraction pa of nests and replace with new

def abandon_nests(nests, pa):
    return [ random_solution() if random < pa else x for x in nests ]
```

Optimize in traffic Mgmt



```
# find and return the best nest

def select_best(nests):
    return min(nests)

# Main loop

while (stopping condition not met):
```

(a) generate new sol^n by levy flights
$x_i' = x_i + \alpha * levy()$
α is scaling factor

(b) evaluate fitness $f(x_i)$

(c) choose a nest randomly, say j among n:
random_sol(n)

(d) If $f(x_i) < f(x_j)$
Replace j with new sol^n of x_i.

(e) A fraction (pa) of worst nests are abandoned and new nests are built at new random locations

(f) Keep the best solutions (nests)

(g) Rank the solutions and find current best nest

end while

Return best sol^n found

Code:

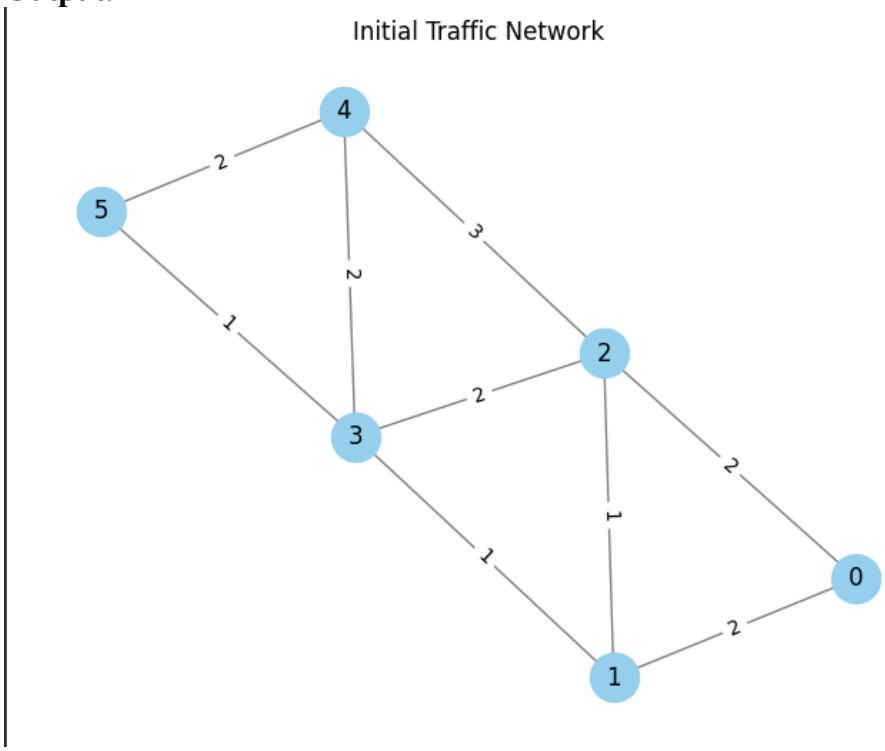
```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random
import math  # Fix: import math for gamma and pi

def create_traffic_graph():
    G = nx.Graph()
    for i in range(6):
        G.add_node(i)
    edges = [
        (0, 1, 2), (0, 2, 2), (1, 2, 1), (1, 3, 1),
        (2, 3, 2), (2, 4, 3), (3, 4, 2), (3, 5, 1),
        (4, 5, 2)
    ]
    for (u, v, dist) in edges:
        G.add_edge(u, v, distance=dist)
    return G

def path_length(G, path):
    length = 0
    for i in range(len(path)-1):
        if G.has_edge(path[i], path[i+1]):
            length += G[path[i]][path[i+1]]['distance']
        else:
            # Invalid path (edge doesn't exist)
            return float('inf')
    return length

def random_path(G, start, end):
    # Generate a random valid path from start to end using DFS
    stack = [(start, [start])]
    paths = []
    while stack:
        (vertex, path) = stack.pop()
        for next_node in set(G.neighbors(vertex)) - set(path):
            if next_node == end:
                paths.append(path + [next_node])
            else:
                stack.append((next_node, path + [next_node]))
        if len(paths) >= 50:  # limit number of paths generated
            break
    if paths:
        return random.choice(paths)
    else:
        # fallback to direct edge if exists
        if G.has_edge(start, end):
```

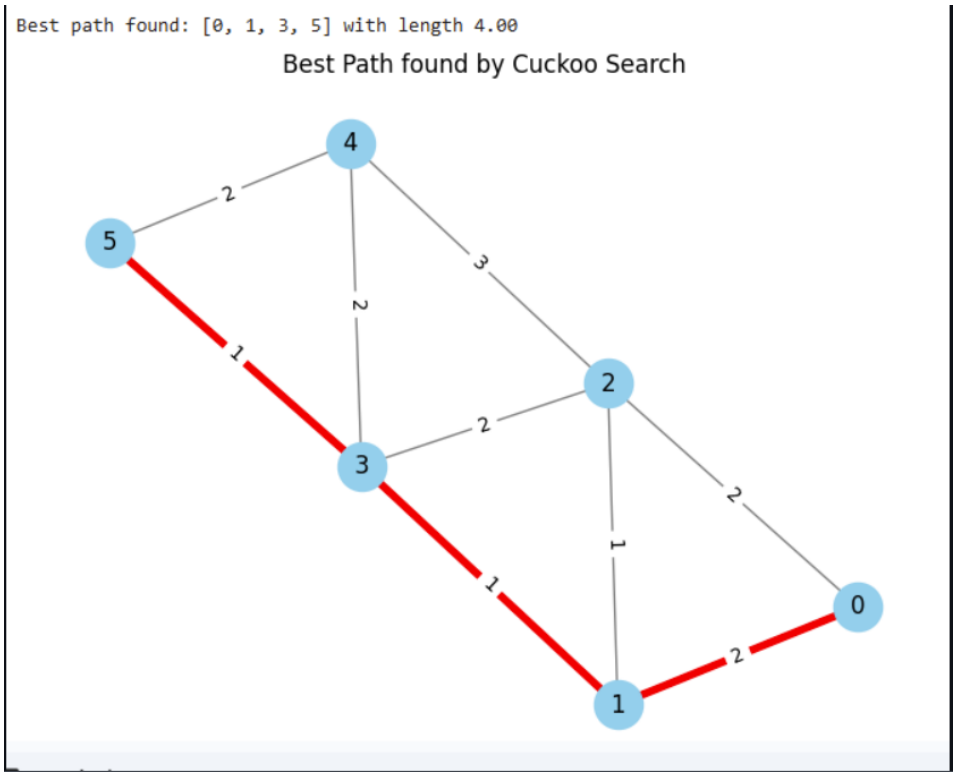
```
            return [start, end]
        else:
            return [start]

def levy_flight(Lambda=1.5):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda /
2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 **
((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.normal(0, sigma_u)
    v = np.random.normal(0, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step

def mutate_path(G, path):
    # Mutate path inspired by Levy flight: randomly remove or
insert nodes
    new_path = path.copy()

    if len(new_path) <= 2:
        # Can't mutate much, just return
        return new_path

    step = int(abs(levy_flight()) * 2)  # scale step for mutation
length
    step = max(1, step)  # Ensure at least 1 step

    # Decide mutation type: insert or remove nodes randomly
    if random.random() < 0.5 and len(new_path) > 2:
        # Remove a segment
        start_idx = random.randint(1, len(new_path) - 2)
        end_idx = min(len(new_path) - 2, start_idx + step)
        # Remove intermediate nodes, reconnect start and end
        if G.has_edge(new_path[start_idx - 1], new_path[end_idx +
1]):
            new_path = new_path[:start_idx] + new_path[end_idx +
1:]
    else:
        # Insert nodes (try to find neighbors to insert)
        insert_pos = random.randint(1, len(new_path) - 1)
        neighbors = list(G.neighbors(new_path[insert_pos - 1]))
        neighbors = [n for n in neighbors if n not in new_path]
        if neighbors:
            insert_node = random.choice(neighbors)
            new_path = new_path[:insert_pos] + [insert_node] +
new_path[insert_pos:]

    # Validate path: if invalid, fallback to original
```

```
    if path_length(G, new_path) == float('inf'):
        return path
    else:
        return new_path

def visualize_graph(G, best_path=None, title="Traffic Network"):
    pos = nx.spring_layout(G, seed=42)
    edge_colors = []
    widths = []
    for u, v in G.edges():
        if best_path and any((u == best_path[i] and v ==
best_path[i+1]) or (v == best_path[i] and u == best_path[i+1]) for
i in range(len(best_path)-1)):
            edge_colors.append('red')
            widths.append(4)
        else:
            edge_colors.append('gray')
            widths.append(1)
    nx.draw(G, pos, with_labels=True, node_color='skyblue',
node_size=600, edge_color=edge_colors, width=widths)
    edge_labels = {(u, v): f"{d['distance']}" for u, v, d in
G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
    plt.title(title)
    plt.show()

def cuckoo_search_traffic(G, start, end, n_nests=15, pa=0.25,
max_iter=100):
    # Initialize nests with random valid paths
    nests = [random_path(G, start, end) for _ in range(n_nests)]
    fitness = [path_length(G, nest) for nest in nests]

    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]

    for iteration in range(max_iter):
        for i in range(n_nests):
            # Generate new solution by mutation (Levy flight
inspired)
            new_nest = mutate_path(G, nests[i])
            new_fitness = path_length(G, new_nest)

            # Replace if new solution is better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness
```

```
                # Update global best
                if new_fitness < best_fitness:
                    best_fitness = new_fitness
                    best_nest = new_nest

        # Abandon a fraction pa of worst nests and generate new
random solutions
        num_abandon = int(pa * n_nests)
        worst_indices = np.argsort(fitness)[-num_abandon:]
        for idx in worst_indices:
            nests[idx] = random_path(G, start, end)
            fitness[idx] = path_length(G, nests[idx])

        print(f"Iteration {iteration+1}/{max_iter}, Best path
length: {best_fitness:.2f}")

    return best_nest, best_fitness

if __name__ == "__main__":
    G = create_traffic_graph()
    visualize_graph(G, title="Initial Traffic Network")

    start_node = 0
    end_node = 5
    best_path, best_length = cuckoo_search_traffic(G, start_node,
end_node, max_iter=50)

    print(f"\nBest path found: {best_path} with length
{best_length:.2f}")
    visualize_graph(G, best_path, title="Best Path found by Cuckoo
Search")
```

Output:

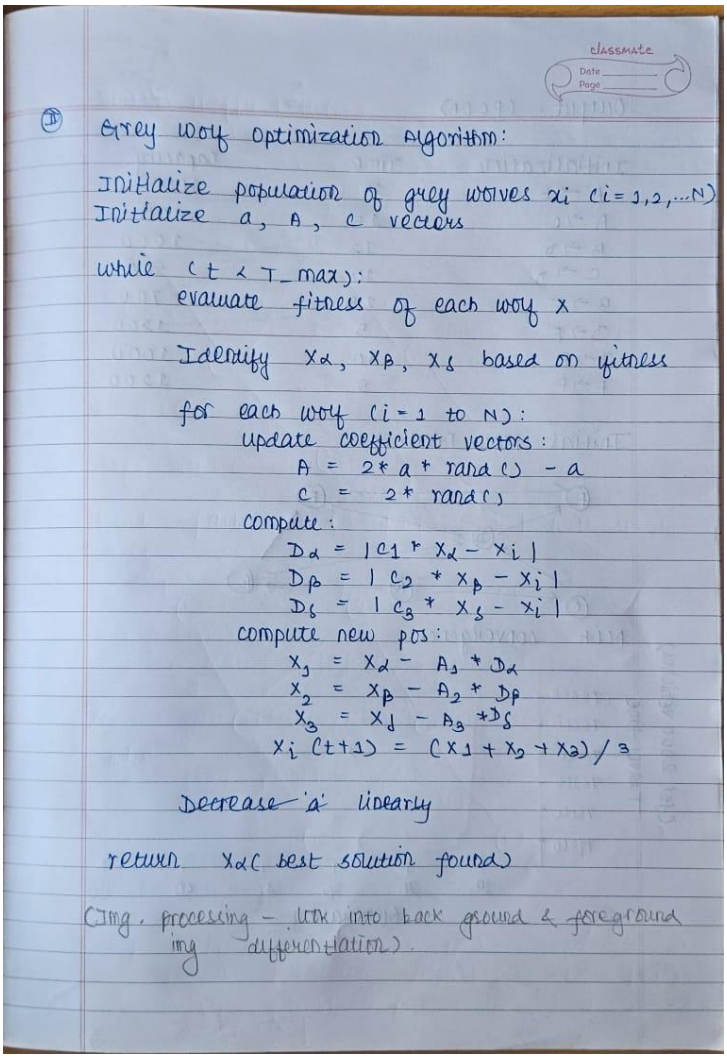

Initial Traffic Network
4
5
2
3
2
0
1
2
3
2
2
1
1
2


Best path found: [0, 1, 3, 5] with length 4.00
Best Path found by Cuckoo Search
4
5
2
3
2
0
1
2
3
2
2
1
1
2

Program 6

Problem statement: Use Grey Wolf Optimization Algorithm to for image processing with respect to background and foreground image differentiation.

Algorithm:



Grey wolf optimization Algorithm:

Initialize population of grey wolves x_i $(i = 1, 2, \ldots N)$
Initialize a, A, C vectors

while (t < T_max):
    evaluate fitness of each wolf X

    Identify X_α, X_β, X_δ based on fitness

    for each wolf (i = 1 to N):
        update coefficient vectors:
            $A = 2 * a * rand() - a$
            $C = 2 * rand()$
        compute:
            $D_\alpha = |C_1 * X_\alpha - X_i|$
            $D_\beta = |C_2 * X_\beta - X_i|$
            $D_\delta = |C_3 * X_\delta - X_i|$
        compute new pos:
            $X_1 = X_\alpha - A_1 * D_\alpha$
            $X_2 = X_\beta - A_2 * D_\beta$
            $X_3 = X_\delta - A_3 * D_\delta$
            $X_i(t+1) = (X_1 + X_2 + X_3)/3$

    Decrease a linearly

return X_α (best solution found)

Img. processing - link into back ground & foreground ing differentiation)

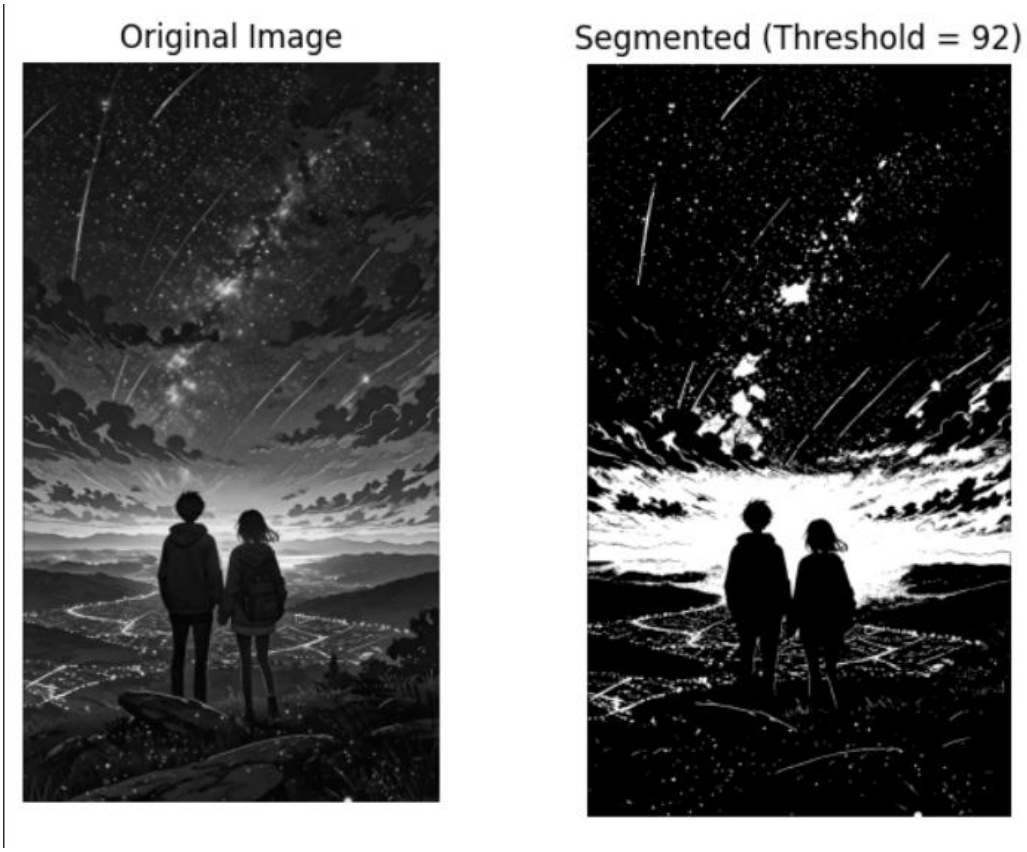
Code:

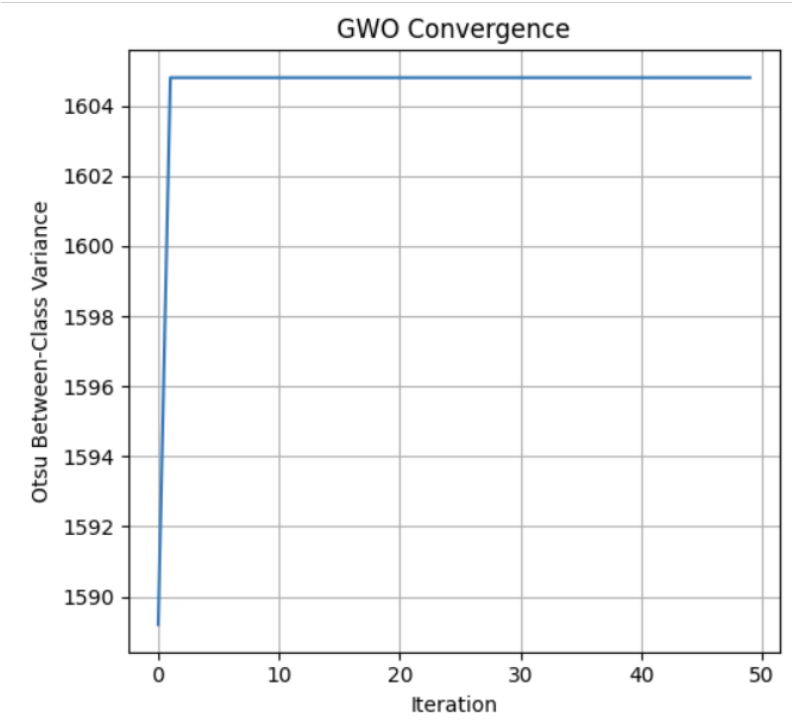
```
# Grey Wolf Optimization (GWO) for Image Thresholding Segmentation
in Colab

# 1. Upload image from your device
from google.colab import files
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print(f"Uploaded image: {img_path}")

# 2. Import required libraries
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random

# 3. Load image and flatten pixel array
img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
if img is None:
    raise FileNotFoundError("Could not read image!")
pixels = img.flatten()

# 4. Fitness Function - Otsu's between-class variance (higher is
better)
def otsu_between_class_variance(threshold, pixels):
    threshold = int(np.clip(threshold, 0, 255))
    hist, _ = np.histogram(pixels, bins=256, range=(0, 256))
    total = pixels.size
    w0 = hist[:threshold].sum() / total
    w1 = hist[threshold:].sum() / total
    if w0 == 0 or w1 == 0:
        return 0.0
    mu0 = np.sum(np.arange(0, threshold) * hist[:threshold]) /
(hist[:threshold].sum() + 1e-8)
    mu1 = np.sum(np.arange(threshold, 256) * hist[threshold:]) /
(hist[threshold:].sum() + 1e-8)
    return w0 * w1 * (mu0 - mu1) ** 2

def fitness(threshold, pixels):
    # GWO minimizes by default, so use negative - higher variance
is better
    return -otsu_between_class_variance(threshold, pixels)

# 5. Minimal GWO Implementation for Single-Threshold Optimization
class GreyWolfOptimizer:
    def __init__(self, n_wolves, n_iters, min_x, max_x):
        self.n_wolves = n_wolves
```

```
        self.n_iters = n_iters
        self.min_x = min_x
        self.max_x = max_x

    def optimize(self, obj_func, pixels):
        wolves = np.random.uniform(self.min_x, self.max_x,
size=(self.n_wolves,))
        alpha_score = float('inf')
        beta_score = float('inf')
        gamma_score = float('inf')
        alpha_pos, beta_pos, gamma_pos = None, None, None

        history = []

        for iter in range(self.n_iters):
            for i in range(self.n_wolves):
                score = obj_func(wolves[i], pixels)
                # Best, second-best, third-best positions
                if score < alpha_score:
                    gamma_score, gamma_pos = beta_score, beta_pos
                    beta_score, beta_pos = alpha_score, alpha_pos
                    alpha_score, alpha_pos = score, wolves[i]
                elif score < beta_score:
                    gamma_score, gamma_pos = beta_score, beta_pos
                    beta_score, beta_pos = score, wolves[i]
                elif score < gamma_score:
                    gamma_score, gamma_pos = score, wolves[i]
            # Position updating
            a = 2 - iter * (2 / self.n_iters)
            for i in range(self.n_wolves):
                for leader_score, leader_pos in zip([alpha_score,
beta_score, gamma_score],
                                                    [alpha_pos,
beta_pos, gamma_pos]):
                    r1, r2 = random.random(), random.random()
                    A = 2 * a * r1 - a
                    C = 2 * r2
                    D = abs(C * leader_pos - wolves[i])
                    X = leader_pos - A * D
                    if leader_pos is alpha_pos:
                        X1 = X
                    elif leader_pos is beta_pos:
                        X2 = X
                    else:
                        X3 = X
                wolves[i] = np.clip((X1 + X2 + X3) / 3, self.min_x,
self.max_x)
            history.append((iter, alpha_pos, alpha_score))
```

```
            print(f"Iter {iter:02d}: best threshold =
{alpha_pos:.2f}, fitness = {alpha_score:.3e}")

        hist_df = pd.DataFrame(history, columns=["iter",
"best_threshold", "fitness"])
        return int(round(alpha_pos)), hist_df

# 6. Run GWO to find the optimal threshold
n_wolves = 20
n_iters = 50
gwo = GreyWolfOptimizer(n_wolves, n_iters, 0, 255)
best_threshold, hist = gwo.optimize(fitness, pixels)
print(f"Best threshold found by GWO: {best_threshold}")

# 7. Segment image and visualize results
_, segmented = cv2.threshold(img, best_threshold, 255,
cv2.THRESH_BINARY)

plt.figure(figsize=(14,5))
plt.subplot(1,3,1)
plt.imshow(img, cmap='gray')
plt.title("Original Image")
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(segmented, cmap='gray')
plt.title(f"Segmented (Threshold = {best_threshold})")
plt.axis('off')

plt.subplot(1,3,3)
plt.plot(hist["iter"], -hist["fitness"])
plt.xlabel("Iteration")
plt.ylabel("Otsu Between-Class Variance")
plt.title("GWO Convergence")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Output:

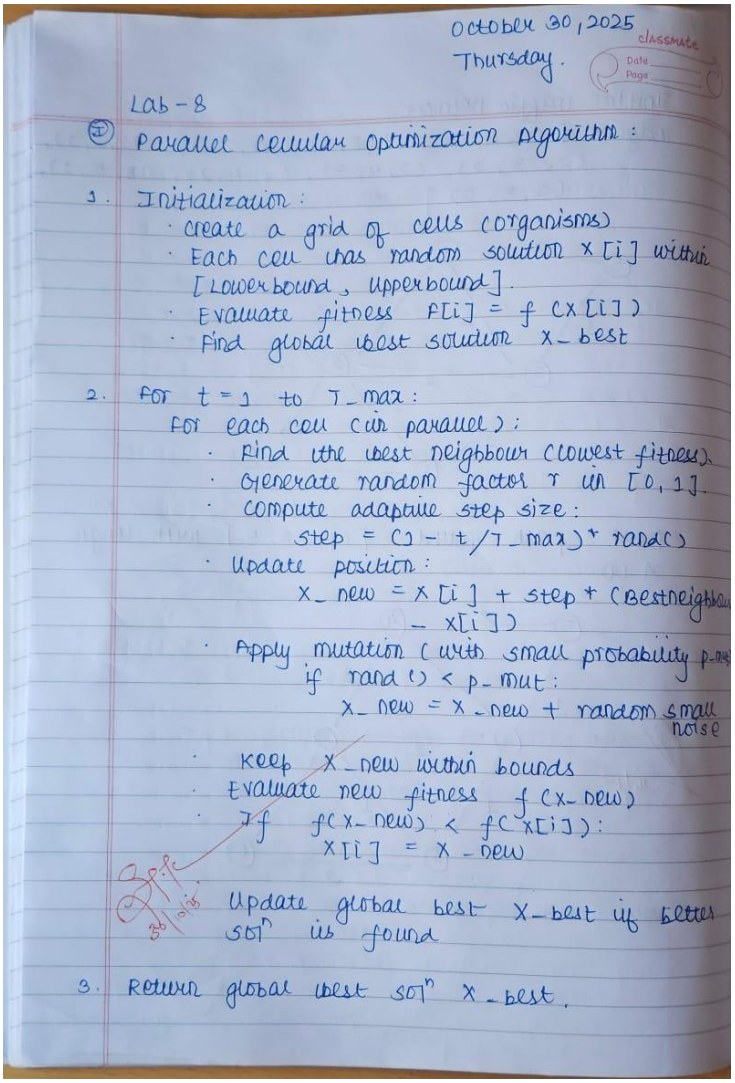

Original Image
Segmented (Threshold = 92)


GWO Convergence
Otsu Between-Class Variance
1604
1602
1600
1598
1596
1594
1592
1590
0
10
20
30
40
50
Iteration

Program 7

Problem statement: Use Parallel cellular-organism Optimization Algorithm for Optimal Traffic Flow Distribution.

Algorithm:



October 30, 2025
Thursday.

Lab - 8

Parallel Cellular Optimization Algorithm :

1. Initialization :
   - Create a grid of cells (organisms)
   - Each cell has random solution X[i] within [Lowerbound, Upperbound].
   - Evaluate fitness F[i] = f(X[i])
   - Find global best solution X_best

2. For t = 1 to T_max :
   For each cell (in parallel) :
   - Find the best neighbour (lowest fitness).
   - Generate random factor r in [0, 1].
   - Compute adaptive step size :
     step = (1 - t/T_max) * rand()
   - Update position :
     X_new = X[i] + step * (Bestneighbour - X[i])
   - Apply mutation (with small probability p_mut)
     if rand() < p_mut :
     X_new = X_new + random small noise
   - Keep X_new within bounds
   - Evaluate new fitness f(X_new)
   - If f(X_new) < f(X[i]) :
     X[i] = X_new
   - Update global best X_best if better sol^n is found

3. Return global best sol^n X_best.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from multiprocessing import Pool
import os
import time
import networkx as nx

# --- 1. Define the Traffic Network ---

TOTAL_VEHICLES = 2000

# BPR Congestion Function Parameters (Standard values)
BPR_ALPHA = 0.15
BPR_BETA = 4.0

EDGES = {
    ('A', 'B'): {'free_time': 10, 'capacity': 1000},
    ('A', 'C'): {'free_time': 15, 'capacity': 800},
    ('B', 'D'): {'free_time': 12, 'capacity': 1000},
    ('C', 'D'): {'free_time': 10, 'capacity': 800},
    ('B', 'E'): {'free_time': 15, 'capacity': 700},
    ('D', 'E'): {'free_time': 5,  'capacity': 1200},
    ('D', 'F'): {'free_time': 10, 'capacity': 1000},
    ('E', 'F'): {'free_time': 8,  'capacity': 1200},
}

PATHS = [
    [('A', 'B'), ('B', 'D'), ('D', 'F')], # Path 0: A->B->D->F
    [('A', 'B'), ('B', 'E'), ('E', 'F')], # Path 1: A->B->E->F
    [('A', 'C'), ('C', 'D'), ('D', 'F')], # Path 2: A->C->D->F
    [('A', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F')]  # Path 3: A-
>C->D->E->F
]
NUM_PATHS = len(PATHS)

# --- 2. PCOA & GA Parameters ---
NUM_CELLS = os.cpu_count() - 1 if os.cpu_count() > 1 else 1
POP_SIZE = 50
GENERATIONS = 75
MIGRATE_EVERY = 10
MUTATION_RATE = 0.2
TOURNAMENT_SIZE = 5

# --- 3. Fitness Function (The Simulation) ---

def calculate_total_travel_time(path_distribution):
    """
```

```
    Fitness function. Calculates total system travel time based on
    a given traffic distribution.
    """
    if not np.isclose(np.sum(path_distribution), 1.0):
        path_distribution = path_distribution /
np.sum(path_distribution)

    edge_volumes = {edge: 0.0 for edge in EDGES}

    for i in range(NUM_PATHS):
        path = PATHS[i]
        volume_on_path = path_distribution[i] * TOTAL_VEHICLES
        for edge in path:
            edge_volumes[edge] += volume_on_path

    total_system_time = 0.0

    for edge, data in EDGES.items():
        volume = edge_volumes[edge]
        capacity = data['capacity']
        free_time = data['free_time']

        if capacity == 0: continue

        congestion_ratio = volume / capacity
        travel_time = free_time * (1 + BPR_ALPHA *
(congestion_ratio ** BPR_BETA))

        total_system_time += travel_time * volume

    if total_system_time == 0:
        return 0.0

    return 1.0 / total_system_time

# --- Helper Function for Network Visualization ---
def get_edge_flows_and_congestion(path_distribution):
    """
    Calculates detailed edge volumes and congestion ratios for
visualization.
    """
    if not np.isclose(np.sum(path_distribution), 1.0):
        path_distribution = path_distribution /
np.sum(path_distribution)

    edge_volumes = {edge: 0.0 for edge in EDGES}
    edge_congestion = {edge: 0.0 for edge in EDGES}
```

```
    for i in range(NUM_PATHS):
        path = PATHS[i]
        volume_on_path = path_distribution[i] * TOTAL_VEHICLES
        for edge in path:
            edge_volumes[edge] += volume_on_path

    for edge, data in EDGES.items():
        volume = edge_volumes[edge]
        capacity = data['capacity']
        if capacity > 0:
            edge_congestion[edge] = volume / capacity
        else:
            edge_congestion[edge] = 0.0

    return edge_volumes, edge_congestion

# --- NEW: Function to draw the network graph ---
def draw_network_graph(title, edge_volumes, edge_congestion, G,
pos, total_vehicles, is_initial=False):
    plt.figure(figsize=(14, 9))

    # Draw nodes
    nx.draw_networkx_nodes(G, pos, node_size=2000,
node_color='lightblue',
                           edgecolors='black', linewidths=1.5)

    # Draw node labels
    nx.draw_networkx_labels(G, pos, font_size=16,
font_weight='bold')

    if is_initial:
        # For initial graph, uniform width and grey color
        edge_widths = [1 for _ in G.edges()] # Uniform width
        edge_colors = ['#cccccc' for _ in G.edges()] # Light grey
        edge_labels = {edge: "" for edge in G.edges()} # No labels
for initial flow
    else:
        # For optimal graph, width and color based on
flow/congestion
        max_flow = max(edge_volumes.values()) if edge_volumes else
1
        edge_widths = [edge_volumes[edge] / (max_flow / 5) + 0.5
for edge in G.edges()] # Scale width
        # RdYlGn_r: Red-Yellow-Green, reversed (0.0=Green, 1.0=Red)
        edge_colors = [plt.cm.RdYlGn_r(min(1.0,
edge_congestion[edge])) for edge in G.edges()]
        edge_labels = {edge: f"{edge_volumes[edge]:.0f}" for edge
in G.edges()}
```

```
    # Draw edges
    nx.draw_networkx_edges(G, pos, edgelist=G.edges(),
                           width=edge_widths,
                           edge_color=edge_colors,
                           node_size=2000,
                           arrowstyle='->',
                           arrowsize=20)

    # Draw edge labels (flow count for optimal graph)
    if not is_initial:
        nx.draw_networkx_edge_labels(G, pos,
edge_labels=edge_labels,
                                     font_color='black',
                                     font_size=10,
                                     bbox=dict(facecolor='white',
alpha=0.5, pad=0.1, edgecolor='none'))

    plt.title(title, fontsize=20)
    if not is_initial:
        plt.text(0, -0.1,
                 'Edge Color = Congestion (Green: Low, Red:
High)\nEdge Width & Label = Vehicle Volume',
                 ha='center', va='center',
transform=plt.gca().transAxes, fontsize=12)
    else:
         plt.text(0, -0.1,
                 'Network structure before optimization. No flow
assigned yet.',
                 ha='center', va='center',
transform=plt.gca().transAxes, fontsize=12)
    plt.axis('off')
    plt.show()

# --- 4. Genetic Algorithm Operations (No Changes) ---
def create_individual():
    rand_arr = np.random.rand(NUM_PATHS)
    return rand_arr / np.sum(rand_arr)

def crossover(parent1, parent2):
    child1 = (parent1 + parent2) / 2.0
    return child1 / np.sum(child1), create_individual()

def mutate(individual):
    if np.random.rand() < MUTATION_RATE:
        idx1, idx2 = np.random.choice(NUM_PATHS, 2, replace=False)
        shift = np.random.rand() * 0.2 * individual[idx1]
        individual[idx1] -= shift
```

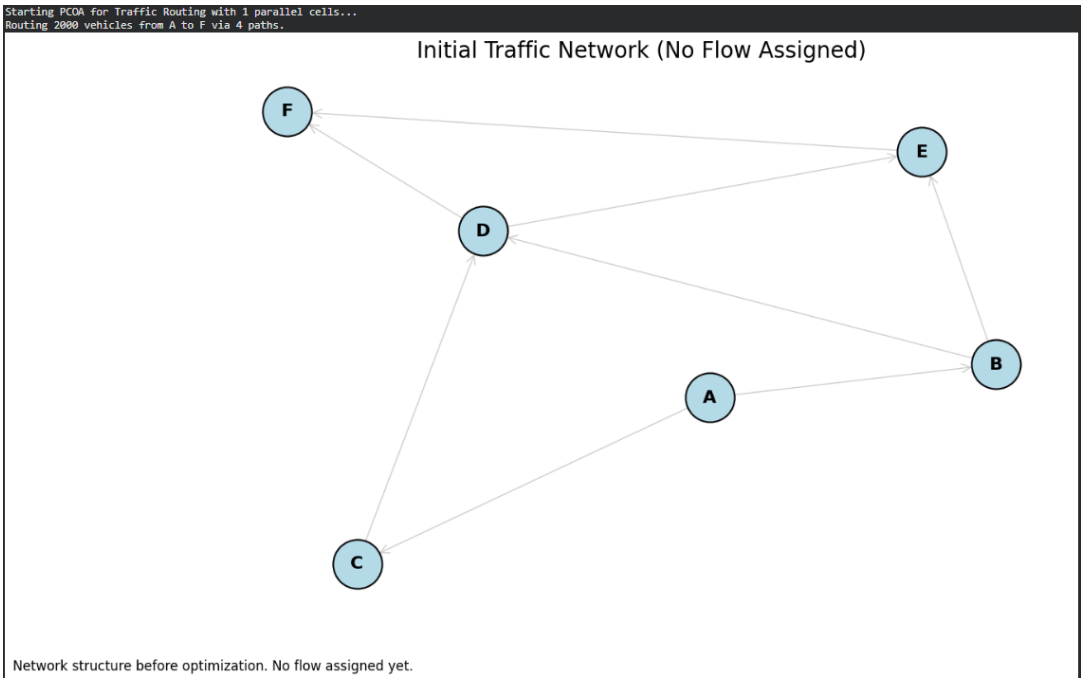
```
            individual[idx2] += shift
            individual = np.clip(individual, 0.0, 1.0)
            individual = individual / np.sum(individual)
    return individual

# --- 5. Parallel Evolution (The 'Cell') (No Changes) ---
def evolve_cell(args):
    population = args[0]
    fitnesses = np.array([calculate_total_travel_time(ind) for ind
in population])

    parents = []
    for _ in range(POP_SIZE // 2):
        competitors_idx = np.random.choice(POP_SIZE,
TOURNAMENT_SIZE, replace=False)
        winner_idx =
competitors_idx[np.argmax(fitnesses[competitors_idx])]
        parents.append(population[winner_idx])

    next_population = []
    best_idx_local = np.argmax(fitnesses)
    best_individual_local = population[best_idx_local]
    next_population.append(best_individual_local) # Elitism

    while len(next_population) < POP_SIZE:
        idx1, idx2 = np.random.choice(len(parents), 2,
replace=False)
        p1, p2 = parents[idx1], parents[idx2]
        c1, c2 = crossover(p1, p2)
        next_population.append(mutate(c1))
        if len(next_population) < POP_SIZE:
            next_population.append(mutate(c2))

    return next_population, (best_individual_local,
fitnesses[best_idx_local])

# --- 6. Main PCOA Orchestrator ---

def run_pcoa_routing_optimization():
    global NUM_CELLS
    print(f"Starting PCOA for Traffic Routing with {NUM_CELLS}
parallel cells...")
    print(f"Routing {TOTAL_VEHICLES} vehicles from A to F via
{NUM_PATHS} paths.")

    # --- Setup for Networkx Graph ---
    G = nx.DiGraph()
    G.add_edges_from(EDGES.keys())
```

```
    pos = nx.spring_layout(G, k=1.5, seed=42) # Fixed positions for
consistent layout

    # --- NEW: Draw the initial network graph ---
    # Dummy values for initial graph (no flow, no congestion yet)
    initial_edge_volumes = {edge: 0.0 for edge in EDGES}
    initial_edge_congestion = {edge: 0.0 for edge in EDGES}
    draw_network_graph('Initial Traffic Network (No Flow
Assigned)',
                       initial_edge_volumes,
initial_edge_congestion,
                       G, pos, TOTAL_VEHICLES, is_initial=True)

    print("\nStarting Optimization Process...\n")

    all_populations = [[create_individual() for _ in
range(POP_SIZE)]
                       for _ in range(NUM_CELLS)]

    best_overall_individual = None
    best_overall_fitness = -np.inf

    best_time_history = []
    avg_time_history = []
    start_time = time.time()

    with Pool(processes=NUM_CELLS) as pool:
        for gen in range(GENERATIONS):
            tasks = [(pop,) for pop in all_populations]
            results = pool.map(evolve_cell, tasks)

            all_populations = [res[0] for res in results]
            gen_bests = [res[1] for res in results]

            all_gen_fitness = [fit for _, fit in gen_bests if fit >
0]
            if not all_gen_fitness: all_gen_fitness = [1e-10]

            best_gen_fitness = max(all_gen_fitness)
            avg_gen_fitness = np.mean(all_gen_fitness)

            best_time_history.append(1.0 / best_gen_fitness)
            avg_time_history.append(1.0 / avg_gen_fitness)

            for ind, fit in gen_bests:
                if fit > best_overall_fitness:
                    best_overall_fitness = fit
                    best_overall_individual = ind
```

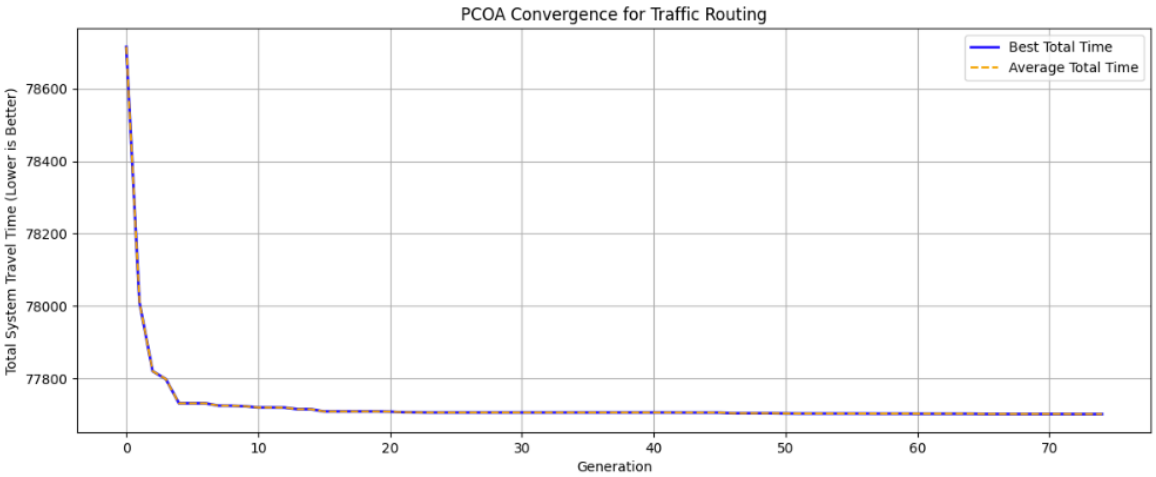
```
            if (gen + 1) % 10 == 0:
                time_val = 1.0 / best_overall_fitness
                print(f"Gen {gen+1:02d}: Min Total Time =
{time_val:.0f} (Vehicle-Minutes)")

            if (gen + 1) % MIGRATE_EVERY == 0 and gen > 0:
                print(f"--- Gen {gen+1}: Migrating best routing
strategies ---")
                best_migrants = [best[0] for best in gen_bests]
                for i in range(NUM_CELLS):
                    migrant = best_migrants[i]
                    target_pop = all_populations[(i + 1) %
NUM_CELLS]
                    fitnesses =
np.array([calculate_total_travel_time(ind) for ind in target_pop])
                    worst_idx = np.argmin(fitnesses)
                    target_pop[worst_idx] = migrant

    end_time = time.time()

    # --- 7. Final Results (Text) ---
    final_time = 1.0 / best_overall_fitness

    print("\n" + "="*40)
    print("🏁 Optimization Finished 🏁")
    print("="*40)
    print(f"Total Optimization Time: {end_time - start_time:.2f}
seconds")
    print(f"**Best Total System Time:** {final_time:.0f} Vehicle-
Minutes")

    print("\nOptimal Routing Distribution:")
    print("-" * 50)
    for i, perc in enumerate(best_overall_individual):
        vehicles_on_path = perc * TOTAL_VEHICLES
        print(f"Path {i}: {perc*100:5.1f}% ({vehicles_on_path:7.0f}
vehicles)")
    print("-" * 50)

    # --- 8. Graphing (Statistics) ---
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

    ax1.plot(best_time_history, label='Best Total Time',
color='blue', linewidth=2)
    ax1.plot(avg_time_history, label='Average Total Time',
color='orange', linestyle='--')
    ax1.set_xlabel('Generation')
```

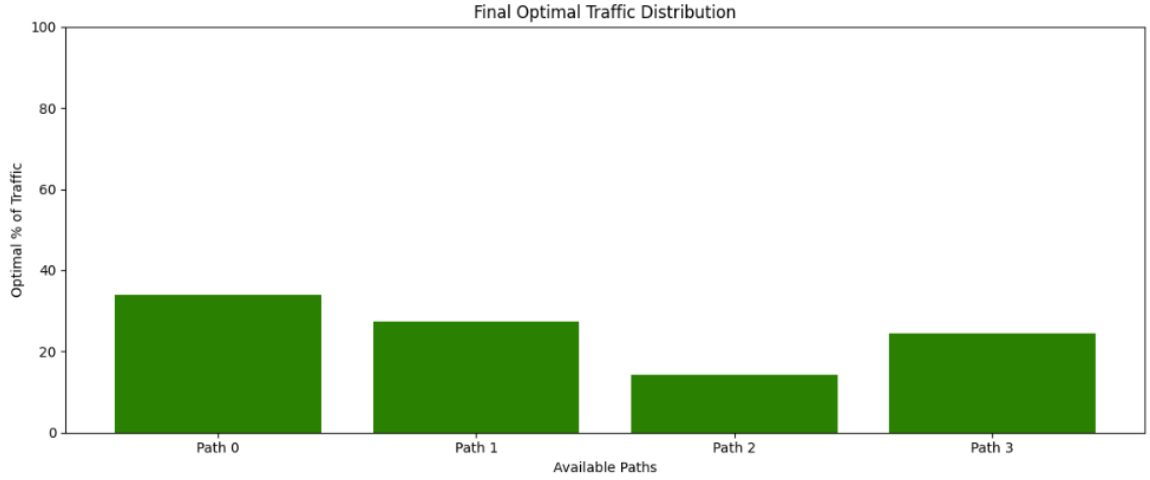
```
    ax1.set_ylabel('Total System Travel Time (Lower is Better)')
    ax1.set_title('PCOA Convergence for Traffic Routing')
    ax1.legend()
    ax1.grid(True)

    path_labels = [f'Path {i}' for i in range(NUM_PATHS)]
    path_percentages = best_overall_individual * 100
    ax2.bar(path_labels, path_percentages, color='green')
    ax2.set_xlabel('Available Paths')
    ax2.set_ylabel('Optimal % of Traffic')
    ax2.set_title('Final Optimal Traffic Distribution')
    ax2.set_ylim(0, 100)

    plt.tight_layout()
    plt.show()

    # --- 9. Draw the final (optimal) network graph ---
    print("\nGenerating final optimal network flow graph...")
    edge_volumes, edge_congestion =
get_edge_flows_and_congestion(best_overall_individual)
    draw_network_graph('Optimal Traffic Flow Distribution',
                       edge_volumes, edge_congestion,
                       G, pos, TOTAL_VEHICLES, is_initial=False)

if __name__ == '__main__':
    run_pcoa_routing_optimization()
```

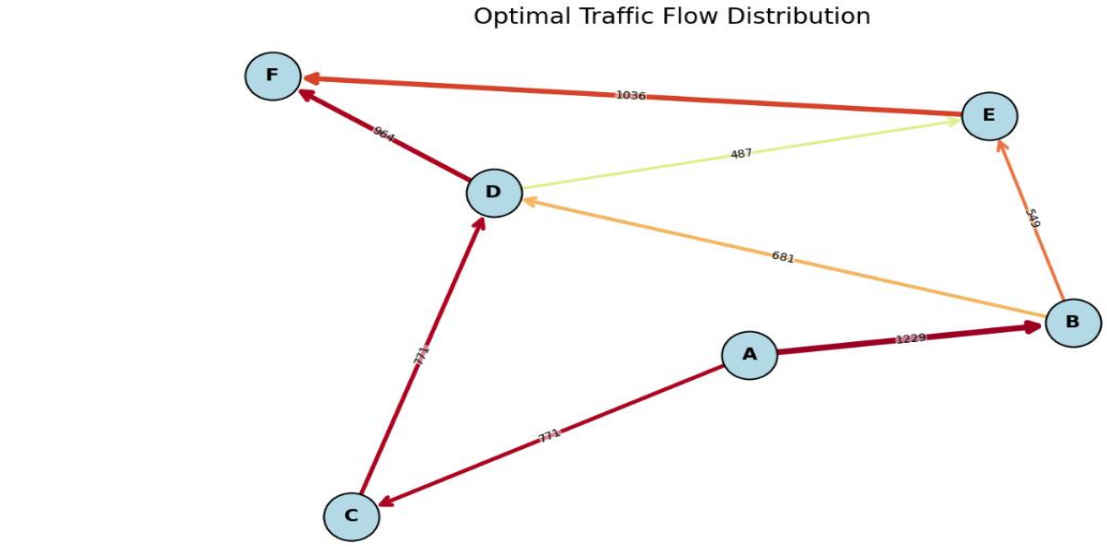
Output:




```
Starting Optimization Process...

Gen 10: Min Total Time = 77723 (Vehicle-Minutes)
--- Gen 10: Migrating best routing strategies ---
Gen 20: Min Total Time = 77709 (Vehicle-Minutes)
--- Gen 20: Migrating best routing strategies ---
Gen 30: Min Total Time = 77706 (Vehicle-Minutes)
--- Gen 30: Migrating best routing strategies ---
Gen 40: Min Total Time = 77706 (Vehicle-Minutes)
--- Gen 40: Migrating best routing strategies ---
Gen 50: Min Total Time = 77704 (Vehicle-Minutes)
--- Gen 50: Migrating best routing strategies ---
Gen 60: Min Total Time = 77702 (Vehicle-Minutes)
--- Gen 60: Migrating best routing strategies ---
Gen 70: Min Total Time = 77702 (Vehicle-Minutes)
--- Gen 70: Migrating best routing strategies ---

========================================
 :  Optimization Finished  :
========================================
Total Optimization Time: 0.73 seconds
**Best Total System Time:** 77701 Vehicle-Minutes

Optimal Routing Distribution:
--------------------------------------------------
Path 0:  34.0% (    681 vehicles)
Path 1:  27.4% (    549 vehicles)
Path 2:  14.2% (    283 vehicles)
Path 3:  24.4% (    487 vehicles)
--------------------------------------------------
```







Edge Color = Congestion (Green: Low, Red: High)
Edge Width & Label = Vehicle Volume