VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Preetham H D (1BM23CS249)

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 **Preetham H D(1BM23CS249),** 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

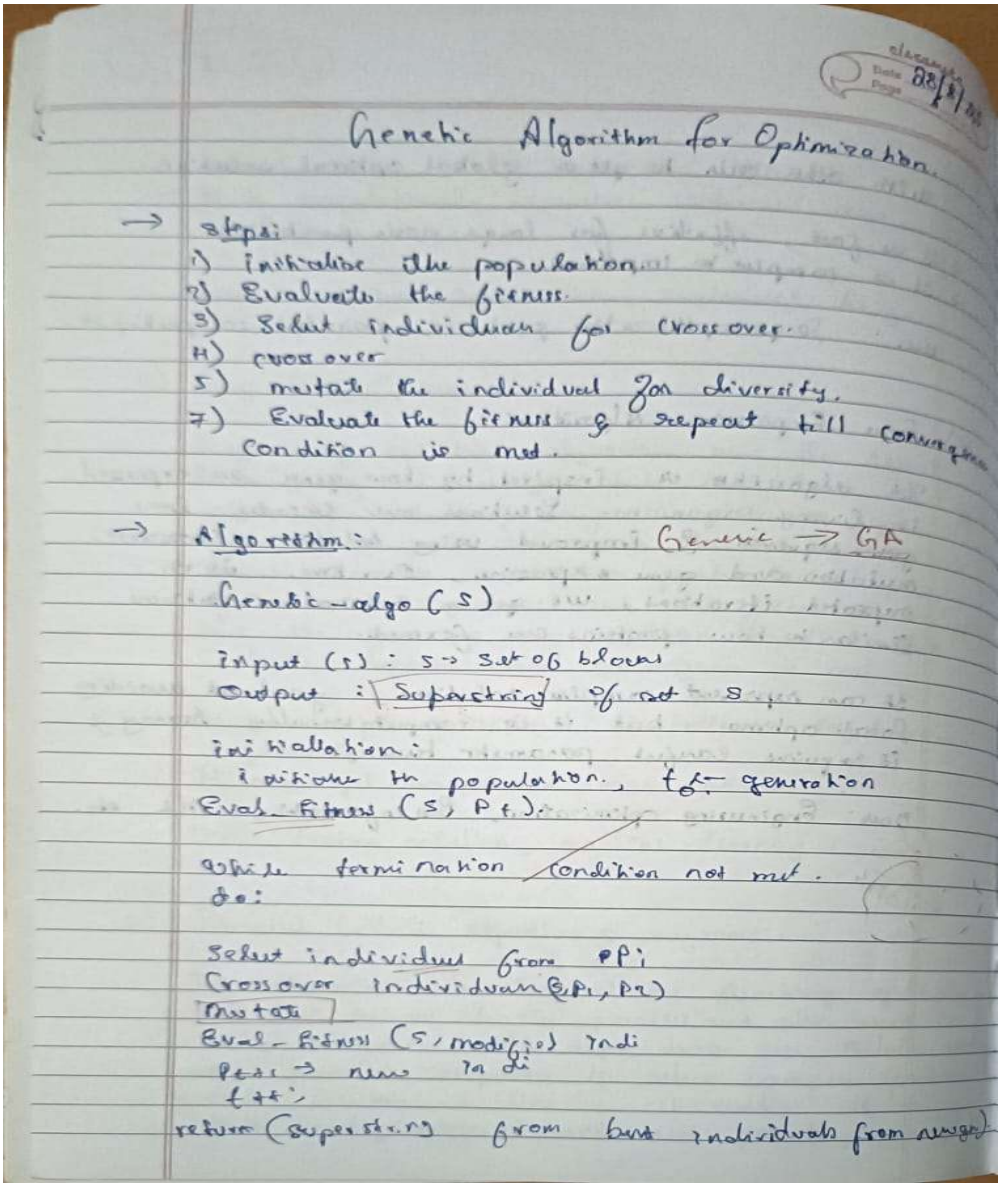
Index

Github Link: https://github.com/Preetham-HD/1BM23CS249-BIS-LAB

Program 1

Genetic Algorithm for Optimization for Neural Networks

Algorithm:



Genetic Algorithm for Optimization.

→ Steps:
1) Initialise the population.
2) Evaluate the fitness.
3) Select individuals for crossover.
4) crossover
5) mutate the individual for diversity.
7) Evaluate the fitness & repeat till convergence condition is met.

→ Algorithm: Generic → GA

Genetic-algo (S)

input (s) : S → set of bloom
Output : Superstring of set S.

initialization:
initialize the population, t_0 ← generation
Eval_fitness (S, P_t).

while termination condition not met.
do:

Select individual from P_i
Crossover individuals(P_1, P_2)
Mutate
Eval_fitness (S, modified) in P_{t+1}
P_{t+1} → new in P_i
t++;
return (superstring) from best individuals from range

mutate (s):
"" s ← target string ""
j : random.random (s);

return j;

⇒ Generic Algorithm.

→ genetic algo (n)
input: n- population size target solution & h.
Output: solution, no of generations ~~(n-typo)~~ iterations.

→ initialise ():
for i in range(N):
population[i] = generate_ini_instance ();

→ fitness ():
for i in population:
eval_fitness ();
while target function not met do:
→ selection ():
Sort the population based on the fitness.
select the individuals from the sorted population.
return p1, p2.

→ crossover ():
~~Selo~~ crossover the p1 & p2 & get a new individual with the new population.
new_popul[i] : crossover (x, y);

mutation ():
mutate the individual in new_popul[i]

→ evaluate the fitness (new-population).

return population[1], gen.;

Execute WRT - Neural N/w s.

Initialize population of N networks with random weights.
For generation = 1 to MAX_GEN:
Evaluate fitness (accuracy) of each network
Select parents based on fitness
perform crossover to create offspring
Apply mutation to offspring
Form new population (elitism + offsprings)
if best fitness >= goal:
Break
Return best neural network.

Output:

Generation 1: Best Accuracy = 0.1600 Avg Accuracy = 0.1028
Generation 2: Best Accuracy = 0.1900 Avg Accuracy: 0.1211
Generation 3: Best Accuracy: 0.1900 Avg Accuracy 0.1154
Generation 4: Best Accuracy: 0.1900 Avg Accuracy: 0.1126
Generation 5: Best Accuracy: 0.1920 Avg Accuracy: 0.118
Generation 6: Best Accuracy: 0.1920 Avg Accuracy: 0.1170
Generation 7: Best Accuracy: 0.1920 Avg Accuracy 0.1170
Generation 8: Best Accuracy: 0.1960 Avg Accuracy 0.1185
Generation 9: Best Accuracy: 0.1970 Avg Accuracy: 0.1185
Generation 10: Best Accuracy: 0.2110 Avg Accuracy: 0.1321
Final training Accuracy: 0.211

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# ========================
# Neural Network
# ========================
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

    def forward(self, X, weights):
        # Decode genome (weights)
        ih_size = self.input_size * self.hidden_size
        ho_size = self.hidden_size * self.output_size

        w_input_hidden = weights[:ih_size].reshape(self.input_size, self.hidden_size)
        w_hidden_output = weights[ih_size:ih_size+ho_size].reshape(self.hidden_size, self.output_size)

        hidden = np.tanh(np.dot(X, w_input_hidden))
        output = self.softmax(np.dot(hidden, w_hidden_output))
        return output

    def softmax(self, z):
        exp = np.exp(z - np.max(z, axis=1, keepdims=True))
        return exp / exp.sum(axis=1, keepdims=True)

# ========================
# Genetic Algorithm
# ========================
class GeneticAlgorithm:
    def __init__(self, nn, pop_size=50, mutation_rate=0.05):
        self.nn = nn
        self.pop_size = pop_size
        self.mutation_rate = mutation_rate

        self.num_weights = (nn.input_size*nn.hidden_size) + (nn.hidden_size*nn.output_size)
        self.population = np.random.uniform(-1, 1, (pop_size, self.num_weights))
        self.history_best = []  # best accuracy per generation
        self.history_avg = []   # average accuracy per generation
```

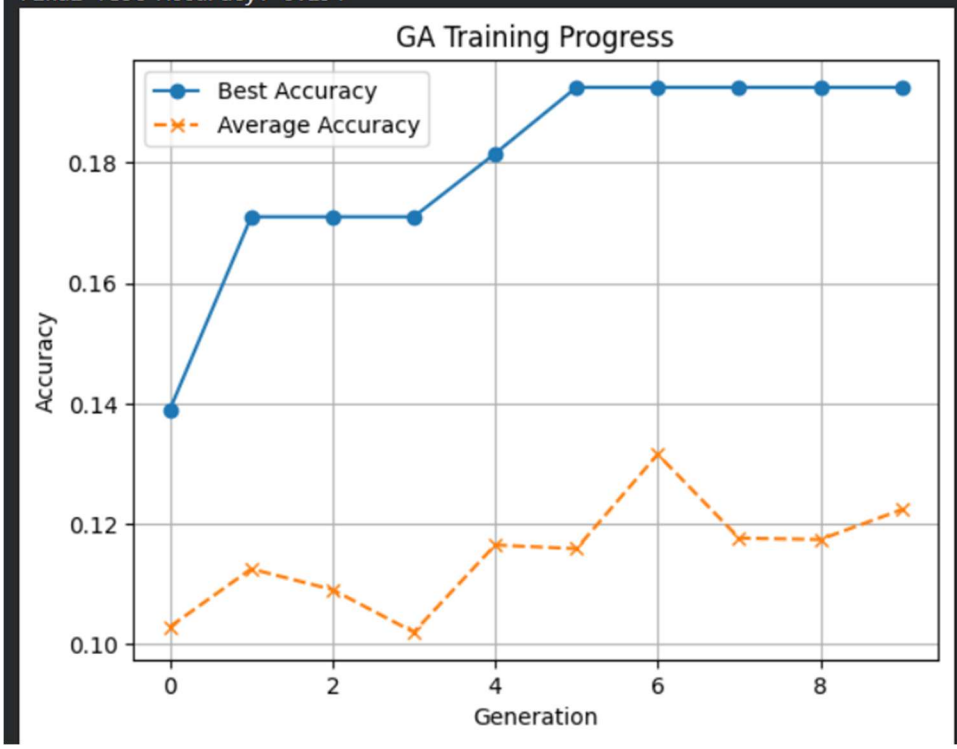
```
    def fitness(self, X, y):
        scores = []
        for individual in self.population:
            preds = self.nn.forward(X, individual)
            acc = np.mean(np.argmax(preds, axis=1) == np.argmax(y, axis=1))
            scores.append(acc)
        return np.array(scores)

    def select(self, fitness_scores):
        probs = fitness_scores / fitness_scores.sum()
        idx = np.random.choice(np.arange(self.pop_size), size=2, p=probs)
        return self.population[idx[0]], self.population[idx[1]]

    def crossover(self, parent1, parent2):
        point = np.random.randint(1, self.num_weights-1)
        child1 = np.concatenate((parent1[:point], parent2[point:]))
        child2 = np.concatenate((parent2[:point], parent1[point:]))
        return child1, child2

    def mutate(self, individual):
        for i in range(len(individual)):
            if np.random.rand() < self.mutation_rate:
                individual[i] += np.random.normal(0, 0.5)
        return individual

    def evolve(self, X, y, generations=20):
        for gen in range(generations):
            fitness_scores = self.fitness(X, y)
            new_population = []

            # Keep elite
            elite_idx = np.argmax(fitness_scores)
            new_population.append(self.population[elite_idx])

            while len(new_population) < self.pop_size:
                p1, p2 = self.select(fitness_scores)
                c1, c2 = self.crossover(p1, p2)
                c1, c2 = self.mutate(c1), self.mutate(c2)
                new_population.extend([c1, c2])

            self.population = np.array(new_population[:self.pop_size])
            best_score = np.max(fitness_scores)
            avg_score = np.mean(fitness_scores)
            self.history_best.append(best_score)
            self.history_avg.append(avg_score)
            print(f"Generation {gen+1}: Best Accuracy = {best_score:.4f}, Avg Accuracy =
{avg_score:.4f}")
```

```
        return self.population[np.argmax(self.fitness(X, y))]

    def plot_progress(self):
        plt.plot(self.history_best, marker='o', label="Best Accuracy")
        plt.plot(self.history_avg, marker='x', linestyle='--', label="Average Accuracy")
        plt.title("GA Training Progress")
        plt.xlabel("Generation")
        plt.ylabel("Accuracy")
        plt.legend()
        plt.grid(True)
        plt.show()

# ========================
# Run GA on MNIST subset
# ========================
if __name__ == "__main__":
    print("Loading MNIST (this may take a minute)...")
    X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
    X = (X / 255.0).astype(np.float32)
    y = y.astype(int)

    # One-hot encode labels
    encoder = OneHotEncoder(sparse_output=False)
    y_onehot = encoder.fit_transform(y.to_numpy().reshape(-1, 1))



    # Train/test split (small subset for demo)
    X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, train_size=2000, test_size=500,
stratify=y, random_state=42)

    nn = NeuralNetwork(input_size=784, hidden_size=64, output_size=10)
    ga = GeneticAlgorithm(nn, pop_size=30, mutation_rate=0.1)
    best_weights = ga.evolve(X_train, y_train, generations=10)

    # Evaluate on training set
    preds_train = nn.forward(X_train, best_weights)
    acc_train = np.mean(np.argmax(preds_train, axis=1) == np.argmax(y_train, axis=1))
    print("Final Training Accuracy:", acc_train)

    # Evaluate on unseen test set
    preds_test = nn.forward(X_test, best_weights)
    acc_test = np.mean(np.argmax(preds_test, axis=1) == np.argmax(y_test, axis=1))
    print("Final Test Accuracy:", acc_test)

    # Plot GA progress (best vs avg accuracy)
```

ga.plot_progress()

Output:

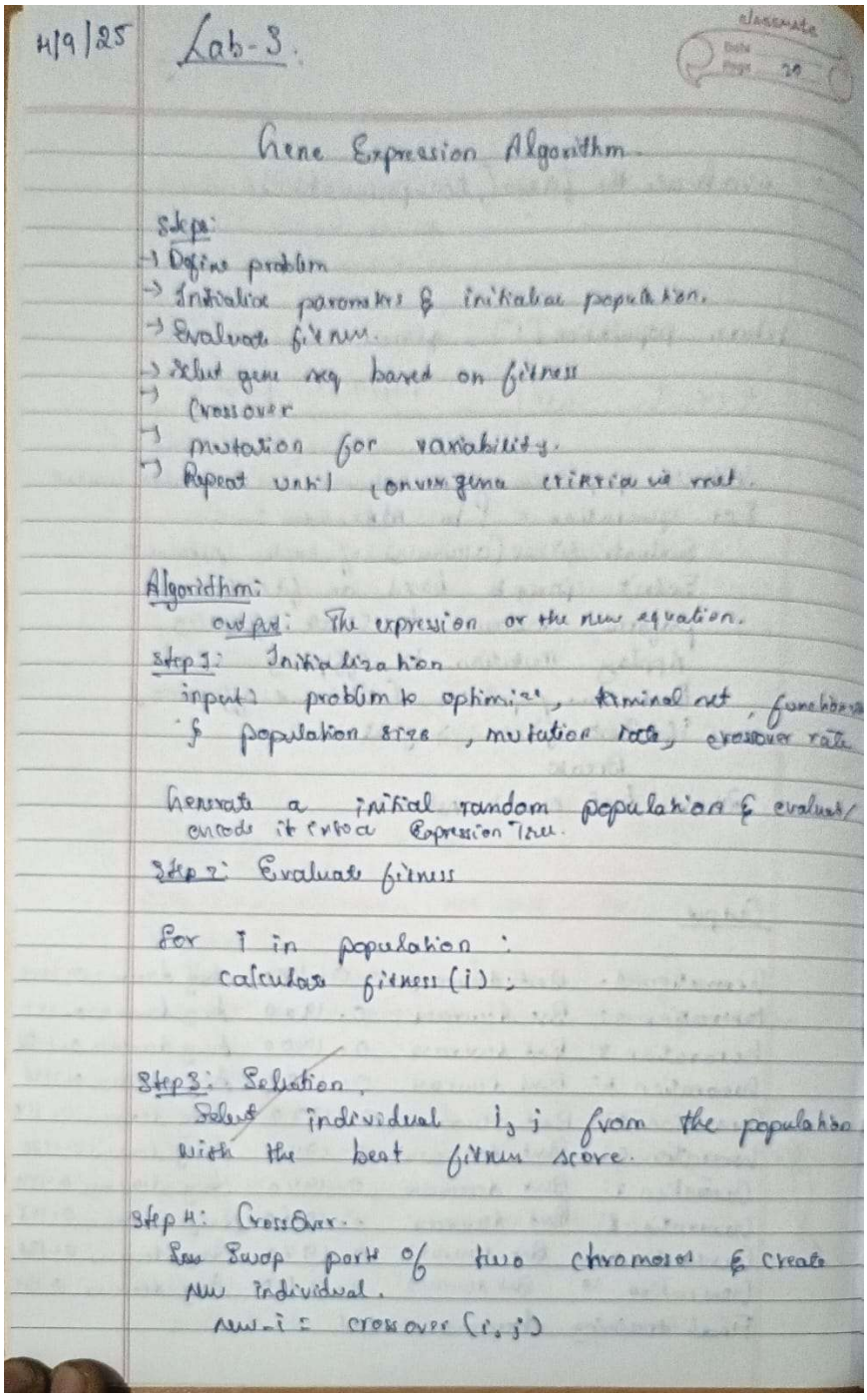
```
Generation 9: Best Accuracy = 0.1925, Avg Accuracy = 0.1174
Generation 10: Best Accuracy = 0.1925, Avg Accuracy = 0.1224
Final Training Accuracy: 0.1925
Final Test Accuracy: 0.194
```



Program 2:

Gene Expression Algorithm for Neural Network

Algorithm:



4/9/25 Lab-3.

Gene Expression Algorithm.

Steps:
→ Define problem
→ Initialize parameters & initial population.
→ Evaluate fitness.
→ Select gen req based on fitness
→ Crossover
→ mutation for variability.
→ Repeat until convergence criteria is met.

Algorithm:
output: The expression or the new equation.
Step 1: Initialization
input: problem to optimize, terminal set, functions & population size, mutation rate, crossover rate.
Generate a initial random population & evaluate/ encode it into a Expression Tree.
Step 2: Evaluate fitness
For I in population :
calculate fitness (i) :
Step 3: Selection.
Select individual i, j from the population with the best fitness score.
Step 4: CrossOver.
Swap parts of two chromosomes & create new individual.
new_i = crossover (i, j)

Step 4: Mutation.

randomly change the value in the new individual.

New_individual[k] = random value;

~~move a substring to another position~~
transpose and inversion of new_individual.

Step5: form new population.

Decode the chromosomes into the population.

New population = new_individual + population[:pop_size]

Step6: Repeat to step 2 to step 5 until the convergence criteria is met.

NN → (GE) → Diff: data

Genetic Algo:

Gen 1, Best Acc = 0.[illegible], Avg Acc = 0.101
Gen 2, Best Acc = 0.201, Avg Acc = 0.101
Gen 3, Best Acc = 0.201, Avg Acc = 0.103
Gen 4, Best Acc = 0.201, Avg Acc = 0.102.
:
Gen 7, Best Acc = 0.203, Avg Acc = 0.101
:
Gen 17, Best Acc = 0.206 Avg Acc = 0.101.
:
Gen 37, Best Acc = 0.209, Avg Acc = 0.105

Code:

```
import numpy as np
import random
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# ============================================================
# 1. Load MNIST subset
# ============================================================
mnist = fetch_openml('mnist_784', version=1)
X = mnist.data.astype(np.float32)[:1000] # use 1000 samples for speed
y = mnist.target.astype(int)[:1000]

scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# ============================================================
# 2. GA + Neural Network
# ============================================================

def softmax(x):
    e = np.exp(x - np.max(x))
    return e / e.sum()

def forward_pass(weights, X, input_size, hidden_size, output_size):
    # unpack weights
    ih_size = input_size * hidden_size
    ho_size = hidden_size * output_size
    w1 = weights[:ih_size].reshape(input_size, hidden_size)
    w2 = weights[ih_size:ih_size+ho_size].reshape(hidden_size, output_size)
    b1 = weights[ih_size+ho_size:ih_size+ho_size+hidden_size]
    b2 = weights[ih_size+ho_size+hidden_size:]
    # forward
    h = np.tanh(X @ w1 + b1)
    o = np.array([softmax(h[i] @ w2 + b2) for i in range(X.shape[0])])
    return o

def accuracy_nn(weights, X, y, input_size, hidden_size, output_size):
    preds = forward_pass(weights, X, input_size, hidden_size, output_size)
    return np.mean(np.argmax(preds, axis=1) == y)
```

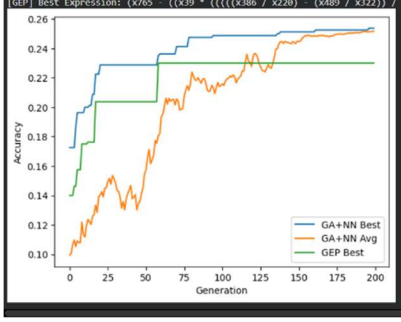
```
def evolve_nn(X_train, y_train, X_test, y_test,
              input_size=784, hidden_size=32, output_size=10,
              pop_size=30, generations=50, mutation_rate=0.1):

    genome_length = input_size*hidden_size + hidden_size*output_size + hidden_size + output_size
    population = [np.random.randn(genome_length) for _ in range(pop_size)]

    best_accs, avg_accs = [], []

    for gen in range(generations):
        scores = [accuracy_nn(ind, X_train, y_train, input_size, hidden_size, output_size) for ind in
population]
        best_idx = np.argmax(scores)
        best_acc, best_ind = scores[best_idx], population[best_idx]
        print(f"[GA+NN] Gen {gen+1}, Best Acc = {best_acc:.3f}, Avg Acc = {np.mean(scores):.3f}")
        best_accs.append(best_acc)
        avg_accs.append(np.mean(scores))

        # next population
        new_pop = [best_ind] # elitism
        while len(new_pop) < pop_size:
            parents = random.sample(population, 2)
            cross_point = random.randint(0, genome_length-1)
            child = np.concatenate([parents[0][:cross_point], parents[1][cross_point:]])
            if random.random() < mutation_rate:
                child[random.randint(0, genome_length-1)] += np.random.randn()
            new_pop.append(child)
        population = new_pop

    final_test_acc = accuracy_nn(best_ind, X_test, y_test, input_size, hidden_size, output_size)
    print("[GA+NN] Final Test Accuracy:", final_test_acc)

    return best_accs, avg_accs

# ============================================================
# 3. Mini GEP (expression trees)
# ============================================================

def add(a, b): return a + b
def sub(a, b): return a - b
def mul(a, b): return a * b
def safe_div(a, b): return a / b if b != 0 else a

FUNCTIONS = [add, sub, mul, safe_div]

def random_expression(num_features, depth=3):
    if depth == 0 or random.random() < 0.3:
```

```
        return ("x", random.randint(0, num_features-1))
    func = random.choice(FUNCTIONS)
    return (func, random_expression(num_features, depth-1), random_expression(num_features,
depth-1))

def eval_expr(expr, sample):
    if expr[0] == "x":
        return sample[expr[1]]
    func, left, right = expr
    return func(eval_expr(left, sample), eval_expr(right, sample))

def fitness(expr, X, y):
    preds = []
    for row in X:
        val = eval_expr(expr, row)
        preds.append(int(val) % 10)
    return accuracy_score(y, preds)

def mutate(expr, num_features):
    if random.random() < 0.2:
        return random_expression(num_features)
    if expr[0] == "x":
        return ("x", random.randint(0, num_features-1))
    func, left, right = expr
    return (func, mutate(left, num_features), mutate(right, num_features))

def crossover(e1, e2):
    return random.choice([e1, e2]) # simple crossover

# ---- Pretty print expression ----
def expr_to_str(expr):
    if expr[0] == "x":
        return f"x{expr[1]}"
    func, left, right = expr
    func_name = {add:"+", sub:"-", mul:"*", safe_div:"/"}[func]
    return f"({expr_to_str(left)} {func_name} {expr_to_str(right)})"

def evolve_gep(X_train, y_train, X_test, y_test, pop_size=20, generations=50):
    population = [random_expression(X_train.shape[1]) for _ in range(pop_size)]
    best_accs = []
    best_expr = None

    for gen in range(generations):
        scored = [(fitness(expr, X_train, y_train), expr) for expr in population]
        scored.sort(key=lambda x: x[0], reverse=True)
        best_fit, best_expr = scored[0]
        best_accs.append(best_fit)
```

```
        print(f"[GEP] Gen {gen+1}, Best Acc = {best_fit:.3f}")

        # next population
        new_pop = [best_expr]
        while len(new_pop) < pop_size:
            parent = random.choice(scored[:5])[1]
            if random.random() < 0.5:
                child = mutate(parent, X_train.shape[1])
            else:
                mate = random.choice(scored[:5])[1]
                child = crossover(parent, mate)
            new_pop.append(child)
        population = new_pop

    final_test_acc = fitness(best_expr, X_test, y_test)
    print("[GEP] Final Test Accuracy:", final_test_acc)
    print("[GEP] Best Expression:", expr_to_str(best_expr))

    return best_accs

# ============================================================
# 4. Run both and compare
# ============================================================

ga_best, ga_avg = evolve_nn(X_train, y_train, X_test, y_test, generations=200)
gep_best = evolve_gep(X_train, y_train, X_test, y_test, generations=200)

# Plot comparison
plt.plot(ga_best, label="GA+NN Best")
plt.plot(ga_avg, label="GA+NN Avg")
plt.plot(gep_best, label="GEP Best")
plt.xlabel("Generation")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Output:

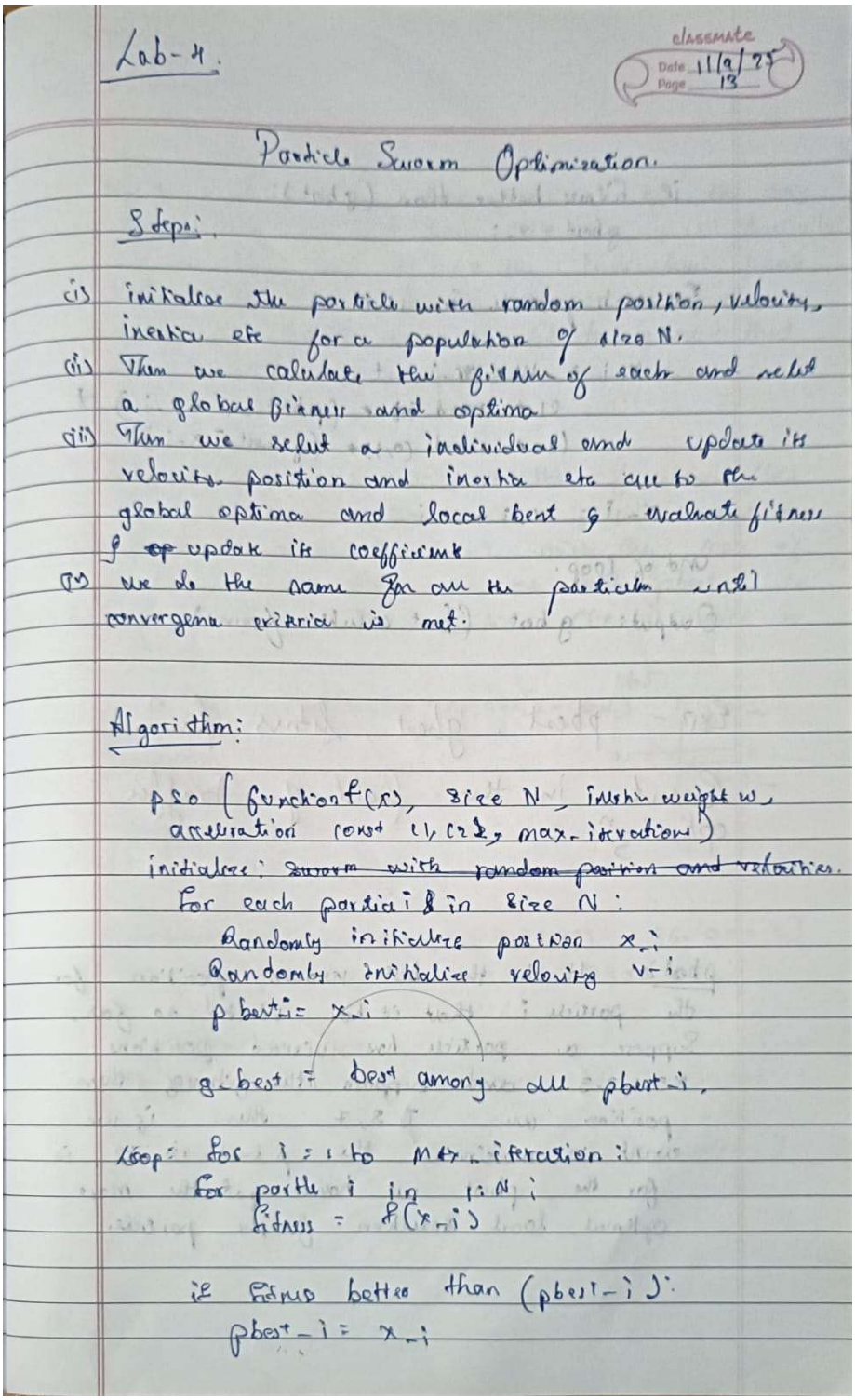
```
[GEP] Gen 180, Best Acc = 0.230
[GEP] Gen 181, Best Acc = 0.230
[GEP] Gen 182, Best Acc = 0.230
[GEP] Gen 183, Best Acc = 0.230
[GEP] Gen 184, Best Acc = 0.230
[GEP] Gen 185, Best Acc = 0.230
[GEP] Gen 186, Best Acc = 0.230
[GEP] Gen 187, Best Acc = 0.230
[GEP] Gen 188, Best Acc = 0.230
[GEP] Gen 189, Best Acc = 0.230
[GEP] Gen 190, Best Acc = 0.230
[GEP] Gen 191, Best Acc = 0.230
[GEP] Gen 192, Best Acc = 0.230
[GEP] Gen 193, Best Acc = 0.230
[GEP] Gen 194, Best Acc = 0.230
[GEP] Gen 195, Best Acc = 0.230
[GEP] Gen 196, Best Acc = 0.230
[GEP] Gen 197, Best Acc = 0.230
[GEP] Gen 198, Best Acc = 0.230
[GEP] Gen 199, Best Acc = 0.230
[GEP] Gen 200, Best Acc = 0.230
[GEP] Final Test Accuracy: 0.195
[GEP] Best Expression: (x765 - ((x39 * (((((x386 / x220) - (x489 / x322)) / ((x277 - x570) - (x278 * x458))) - ((x387 * (x324 - x187)) - ((x373 / x266) / x617))) + (((x175 * x203) * (x211 * x599))
```



Program 3:

Particle Swarm Optimization for Traffic Optimization

Algorithm:



Lab-4.

classmate
Date 11/9/25
Page 13

Particle Swarm Optimization.

Steps:

(i) initialise the particle with random position, velocity, inertia etc for a population of size N.
(ii) Then we calculate the fitness of each and select a global finest and optima
(iii) Then we select a individual and update its velocity position and inertia etc all to the global optima and local best & evaluate fitness & ~~op~~ update its coefficient
(iv) we do the same for all the particles until convergence criteria is met.

Algorithm:

pso (function f(x), size N, inertia weight w, acceleration const c1, c2, max_iteration)
initialise: ~~swarm with random position and velocities.~~
for each particle i in size N:
Randomly initialize position x_i
Randomly initialize velocity v_i
p best_i = x_i

g_best = best among all pbest_i.

Loop: for i = 1 to max_iteration:
for particle i in 1:N:
fitness = f(x_i)

if fitness better than (pbest_i):
pbest_i = x_i

Code:

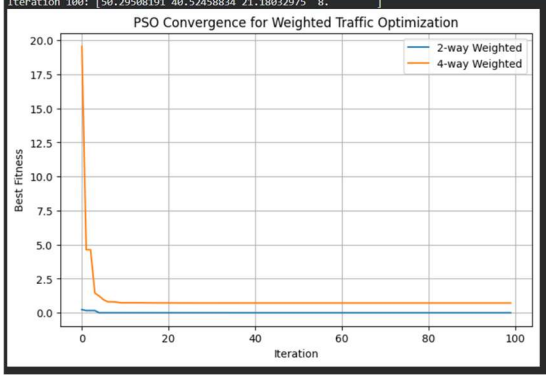
```
import numpy as np
import matplotlib.pyplot as plt

# ----------------------------
# Setup
# ----------------------------
CYCLE_TIME = 120

# Weighted Fitness Functions
def fitness_two_weighted(x, weights=[0.7, 0.3]):
    """2-way: Weighted imbalance penalty (heavier traffic on NS)."""
    t_ns, t_ew = x
    ideal = np.array([CYCLE_TIME * 0.7, CYCLE_TIME * 0.3])
    wait = np.sum(weights * np.abs(x - ideal) ** 1.5)
    return wait

def fitness_four_weighted(x, weights=[0.4, 0.3, 0.2, 0.1]):
    """4-way: Weighted imbalance penalty (heavier traffic on NS straight)."""
    ideal = np.array([50, 40, 20, 10]) # based on demand
    wait = np.sum(weights * np.abs(x - ideal) ** 1.5)
    return wait


# ----------------------------
# Generic PSO Implementation
# ----------------------------
def PSO(fitness_func, dim, num_particles=20, max_iter=100,
        w=0.7, c1=1.5, c2=1.5, min_time=8):

    # Initialize random particles
    particles = np.random.rand(num_particles, dim) * CYCLE_TIME
    particles = np.array([p / sum(p) * CYCLE_TIME for p in particles])
    velocities = np.random.randn(num_particles, dim)

    # Initialize bests
    pbest_positions = particles.copy()
    pbest_scores = np.array([fitness_func(p) for p in particles])
    gbest_index = np.argmin(pbest_scores)
    gbest_position = pbest_positions[gbest_index].copy()
```

```
gbest_score = pbest_scores[gbest_index]

convergence = []
best_positions_history = [] # Store best positions at each iteration

for _ in range(max_iter):
    for i in range(num_particles):
        score = fitness_func(particles[i])

        if score < pbest_scores[i]:
            pbest_scores[i] = score
            pbest_positions[i] = particles[i].copy()

        if score < gbest_score:
            gbest_score = score
            gbest_position = particles[i].copy()

    for i in range(num_particles):
        r1, r2 = np.random.rand(), np.random.rand()
        velocities[i] = (
            w * velocities[i]
            + c1 * r1 * (pbest_positions[i] - particles[i])
            + c2 * r2 * (gbest_position - particles[i])
        )
        particles[i] += velocities[i]

        # Normalize to cycle time
        total = np.sum(particles[i])
        particles[i] = particles[i] / total * CYCLE_TIME

        # Clip min green time
        particles[i] = np.clip(particles[i], min_time, CYCLE_TIME - min_time)
        total = np.sum(particles[i])
        particles[i] = particles[i] / total * CYCLE_TIME

    convergence.append(gbest_score)
    best_positions_history.append(gbest_position.copy())

return gbest_position, gbest_score, convergence, best_positions_history
```

```
# ----------------------------
# Run PSO for 2-way intersection (weighted)
# ----------------------------
best_two, score_two, conv_two, history_two = PSO(fitness_two_weighted, dim=2)
print("=== 2-Way Weighted Traffic Optimization ===")
print("Optimal Signal Timings:", best_two)
print("Best Fitness (waiting cost):", score_two)
print("\nSignal Timings History (2-way):")
for i, pos in enumerate(history_two):
    print(f"Iteration {i+1}: {pos}")


# ----------------------------
# Run PSO for 4-way intersection (weighted)
# ----------------------------
best_four, score_four, conv_four, history_four = PSO(fitness_four_weighted, dim=4)
print("\n=== 4-Way Weighted Traffic Optimization ===")
print("Optimal Signal Timings:", best_four)
print("Best Fitness (waiting cost):", score_four)
print("\nSignal Timings History (4-way):")
for i, pos in enumerate(history_four):
    print(f"Iteration {i+1}: {pos}")


# ----------------------------
# Plot convergence
# ----------------------------
plt.figure(figsize=(8,5))
plt.plot(conv_two, label="2-way Weighted")
plt.plot(conv_four, label="4-way Weighted")
plt.xlabel("Iteration")
plt.ylabel("Best Fitness")
plt.title("PSO Convergence for Weighted Traffic Optimization")
plt.legend()
plt.grid(True)
plt.show()
```

Output:

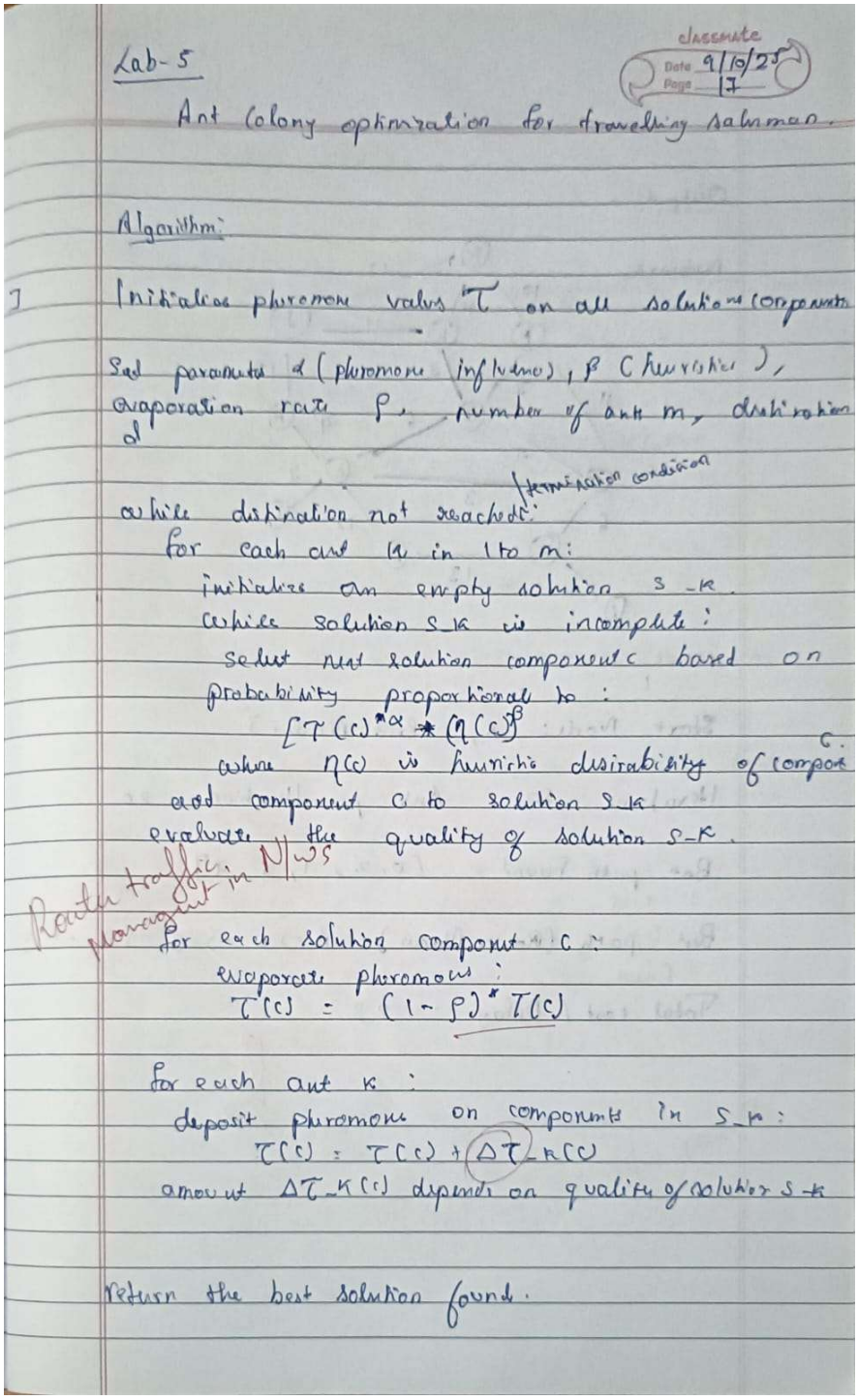
```
Iteration 79: [50.29502899 40.52451952 21.1804515   8.        ]
Iteration 80: [50.29505333 40.52461114 21.18033553  8.        ]
Iteration 81: [50.29505333 40.52461114 21.18033553  8.        ]
Iteration 82: [50.29508088 40.52457612 21.180343    8.        ]
Iteration 83: [50.29508088 40.52457612 21.180343    8.        ]
Iteration 84: [50.29508088 40.52457612 21.180343    8.        ]
Iteration 85: [50.29508088 40.52457612 21.180343    8.        ]
Iteration 86: [50.2950826  40.52457752 21.18033988  8.        ]
Iteration 87: [50.2950826  40.52457752 21.18033988  8.        ]
Iteration 88: [50.29508727 40.52458134 21.18033139  8.        ]
Iteration 89: [50.29508727 40.52458134 21.18033139  8.        ]
Iteration 90: [50.29508608 40.52458037 21.18033355  8.        ]
Iteration 91: [50.29508608 40.52458037 21.18033355  8.        ]
Iteration 92: [50.29508632 40.52458057 21.18033312  8.        ]
Iteration 93: [50.29508632 40.52458057 21.18033312  8.        ]
Iteration 94: [50.2950828  40.52458485 21.18033234  8.        ]
Iteration 95: [50.2950828  40.52458485 21.18033234  8.        ]
Iteration 96: [50.2950828  40.52458485 21.18033234  8.        ]
Iteration 97: [50.29508555 40.52458678 21.18032767  8.        ]
Iteration 98: [50.29508341 40.52458769 21.1803289   8.        ]
Iteration 99: [50.29508191 40.52458834 21.18032975  8.        ]
Iteration 100: [50.29508191 40.52458834 21.18032975  8.        ]
```



Program 4:

Ant colony Optimization for Router traffic management

Algorithm:



Lab-5

Ant Colony optimization for travelling salesman.

Algorithm:

Initialize pheromone value τ on all solution components

Set parameter α (pheromone influence), β (heuristic), evaporation rate ρ, number of ants m, destination d

while destination not reached: (termination condition)

for each ant k in 1 to m:

initialize an empty solution S_k.

while solution S_k is incomplete:

select next solution component c based on probability proportional to:

$[\tau(c)]^{\alpha} * (\eta(c))^{\beta}$

where $\eta(c)$ is heuristic desirability of component c.

add component c to solution S_k

evaluate the quality of solution S_k.

Router traffic management in NWS

for each solution component c:

evaporate pheromone:

$\tau(c) = (1 - \rho) * \tau(c)$

for each ant k:

deposit pheromone on components in S_k:

$\tau(c) = \tau(c) + \Delta\tau_k(c)$

amount $\Delta\tau_k(c)$ depends on quality of solution S_k

return the best solution found.

Code:

```
import numpy as np
import random

class ACO_Router:

    def __init__(self, cost_matrix, n_ants=10, n_iterations=100, alpha=1.0, beta=5.0, rho=0.5, Q=1.0):

        self.cost_matrix = np.array(cost_matrix)
        self.n_nodes = self.cost_matrix.shape[0]


        self.n_ants = n_ants # no of ants
        self.n_iterations = n_iterations #no of cycles
        self.alpha = alpha # Pheromone
        self.beta = beta # Heuristic
        self.rho = rho # Pheromone evaporation rate
        self.Q = Q # Pheromone deposit constant


        self.pheromone = np.ones((self.n_nodes, self.n_nodes)) / self.n_nodes


        self.best_path = None
        self.best_cost = np.inf

    def _get_heuristic(self, i, j):
        return 1.0 / (self.cost_matrix[i, j] + 1e-9)

    def _choose_next_node(self, current_node, visited):
        unvisited = [j for j in range(self.n_nodes) if j not in visited and self.cost_matrix[current_node, j]
> 0]

        if not unvisited:
            return None


        probabilities = []
        for j in unvisited:
            tau = self.pheromone[current_node, j] # Pheromone
```

```
            eta = self._get_heuristic(current_node, j) # Heuristic


            probabilities.append((tau ** self.alpha) * (eta ** self.beta))


        total_prob = sum(probabilities)
        if total_prob == 0:

            return random.choice(unvisited)

        normalized_probs = [p / total_prob for p in probabilities]


        next_node = random.choices(unvisited, weights=normalized_probs, k=1)[0]
        return next_node

    def _run_ant(self, start_node, end_node):

        path = [start_node]
        current_node = start_node
        path_cost = 0

        while current_node != end_node:
            next_node = self._choose_next_node(current_node, path)

            if next_node is None:

                return None, np.inf

            path_cost += self.cost_matrix[current_node, next_node]
            path.append(next_node)
            current_node = next_node


            if len(path) > self.n_nodes:
                return None, np.inf

        return path, path_cost

    def _update_pheromone(self, ant_paths):
```

```
        self.pheromone *= (1 - self.rho)


        for path, cost in ant_paths:
            if path is not None and cost < np.inf:
                deposit = self.Q / cost
                for i in range(len(path) - 1):
                    start, end = path[i], path[i+1]
                    self.pheromone[start, end] += deposit

    def find_shortest_path(self, start_node, end_node):
        for iteration in range(self.n_iterations):
            ant_paths = []

            for _ in range(self.n_ants):
                path, cost = self._run_ant(start_node, end_node)
                ant_paths.append((path, cost))

                if cost < self.best_cost:
                    self.best_cost = cost
                    self.best_path = path

            self._update_pheromone(ant_paths)

        return self.best_path, self.best_cost

COST_MATRIX = [
    [ 0, 7, 3, 0, 6, 11, 0, 0, 0, 0, 0, 0],
   [ 7, 0, 6, 0, 0, 7, 3, 0, 0, 0, 0, 0],
    [ 3, 6, 0, 12, 0, 0, 0, 9, 7, 0, 0, 0],
    [ 0, 0, 12, 0, 13, 0, 0, 0, 0, 0, 0, 0],
    [ 6, 0, 0, 13, 0, 0, 0, 0, 0, 0, 0, 11],
    [11, 7, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0],
    [ 0, 3, 0, 0, 0, 0, 0, 0, 0, 11, 11, 0],
    [ 0, 0, 9, 0, 0, 0, 0, 0, 0, 18, 0, 0],
    [ 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [ 0, 0, 0, 0, 0, 0, 11, 18, 0, 0, 13, 0],
    [ 0, 0, 0, 0, 0, 12, 11, 0, 0, 13, 0, 0],
    [ 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 0, 0]
]
```

```
START_NODE = 0
END_NODE = 9

aco = ACO_Router(
    cost_matrix=COST_MATRIX,
    n_ants=20,
    n_iterations=50,
    alpha=1.0,
    beta=5.0,
    rho=0.1,
    Q=10.0
)

shortest_path, cost = aco.find_shortest_path(START_NODE, END_NODE)


node_map = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E',5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K'}
path_names = [node_map[n] for n in shortest_path] if shortest_path else "None"

print("--- ACO Router Traffic Management Simulation ---")
print(f"Start Node: {node_map[START_NODE]}, End Node: {node_map[END_NODE]}")
print(f"Iterations: {aco.n_iterations}, Ants per Iteration: {aco.n_ants}\n")

if shortest_path:
    print(f"Best Path Found (Node Indices): {shortest_path}")
    print(f"Best Path (Router Names): {' -> '.join(path_names)}")
    print(f"Total Cost/Delay: {cost}")
else:
    print("No path found.")
```

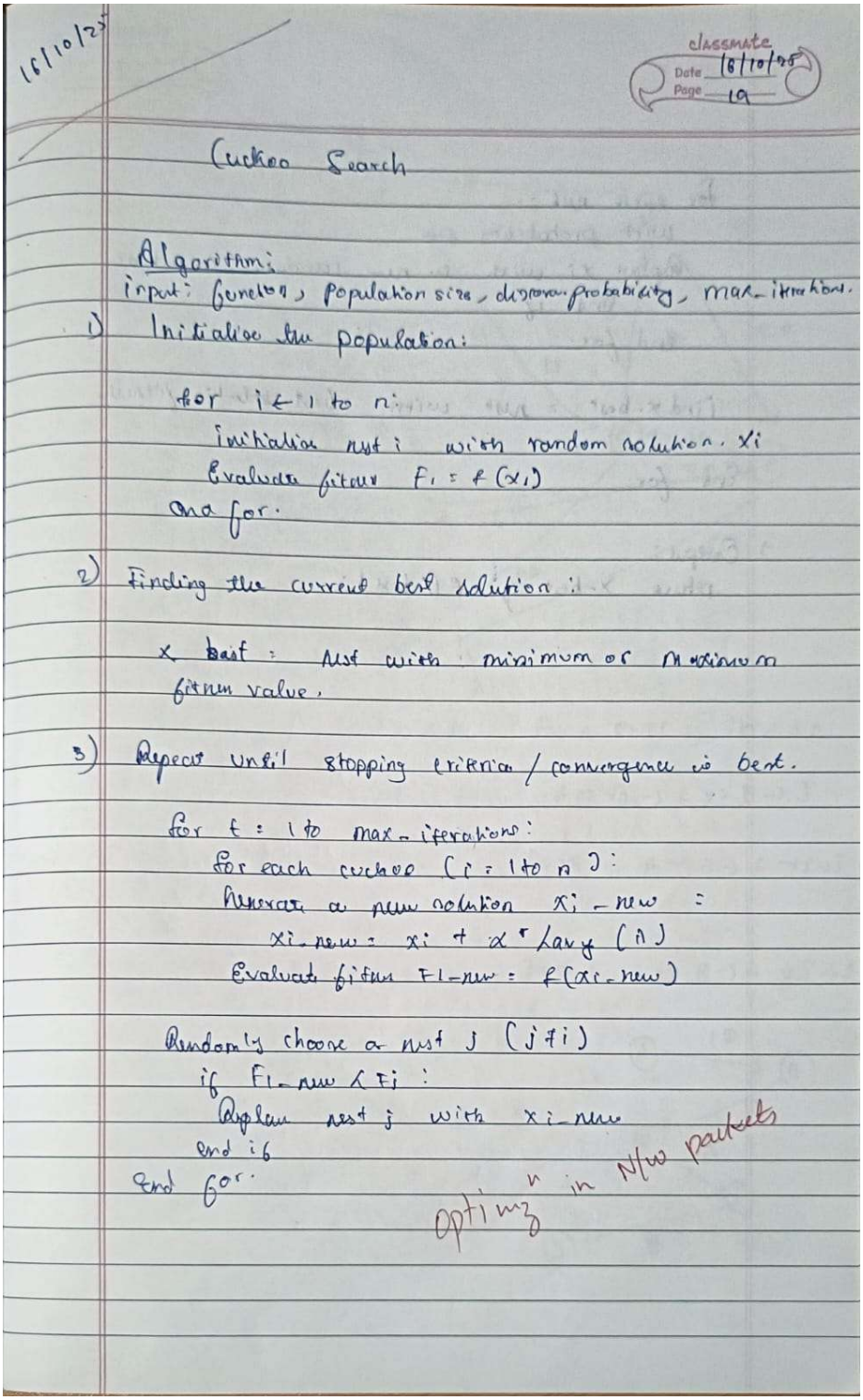
Output:

```
--- ACO Router Traffic Management Simulation ---
Start Node: A, End Node: J
Iterations: 50, Ants per Iteration: 20

Best Path Found (Node Indices): [0, 2, 1, 6, 9]
Best Path (Router Names): A -> C -> B -> G -> J
Total Cost/Delay: 23
```

Program 5:

Cuckoo Search for Optimization in Network Packets

Algorithm:



Cuckoo Search

Algorithm:
input: function, population size, discovery probability, max_iterations.

1) Initialise the population:

for i ← 1 to n:
Initialise nest i with random solution xi
Evaluate fitness Fi = f(xi)
end for.

2) Finding the current best solution:

x best : Nest with minimum or maximum fitness value.

3) Repeat until stopping criteria / convergence is best.

for t = 1 to max_iterations:
for each cuckoo (i = 1 to n):
Generate a new solution xi_new =
xi_new = xi + α * Levy(λ)
Evaluate fitness Fi_new = f(xi_new)

Randomly choose a nest j (j ≠ i)
if Fi_new < Fj:
Replace nest j with xi_new
end if
end for.

optimization in N/w packets

for each nest i:
with probability pa:
Replace xi with a new random solution.
end if
End for.

Find x-best = nest with best ~~solution~~ fitness.

End for.

→ Output:
return x-best $f(x\text{-best})$;

Code:

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import random
import math

# --- 1. Graph Setup ---
def create_complex_graph():
    """
    Creates a complex, directed graph representing a network.
    Edges have 'cost' attributes for optimization.
    """
    G = nx.DiGraph()

    # Nodes (e.g., routers)
    nodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
    G.add_nodes_from(nodes)

    # Edges with costs (e.g., latency in ms)
    # (source, target, cost)
    edges = [
        ('A', 'B', 10), ('A', 'C', 15), ('A', 'D', 25),
        ('B', 'E', 20), ('B', 'F', 5),
        ('C', 'E', 10), ('C', 'G', 40),
        ('D', 'F', 30), ('D', 'H', 15),
        ('E', 'I', 10), ('E', 'G', 5),
        ('F', 'I', 15), ('F', 'H', 20),
        ('G', 'J', 10),
        ('H', 'J', 12),
        ('I', 'J', 8),
        # Backwards/Alternative paths for complexity
        ('B', 'A', 12), ('C', 'B', 5), ('E', 'C', 15),
        ('H', 'E', 25), ('I', 'G', 15)
    ]

    for u, v, cost in edges:
        G.add_edge(u, v, cost=cost)
```

```
    return G

# --- 2. Path Encoding and Fitness ---

def calculate_path_cost(G, path):
    """
    Calculates the total cost (fitness) of a path.
    A path is a list of nodes, e.g., ['A', 'B', 'E', 'I', 'J'].
    Returns a very high cost for invalid paths.
    """
    if not path or len(path) < 2:
        return float('inf')

    total_cost = 0
    for i in range(len(path) - 1):
        u, v = path[i], path[i+1]
        if G.has_edge(u, v):
            total_cost += G[u][v]['cost']
        else:
            # High cost for an invalid hop (broken link)
            return float('inf')

    return total_cost

def generate_random_valid_path(G, source, destination, max_length=10):
    """
    Generates a random, valid path between source and destination.
    Uses a simple random walk until the destination is reached or max_length is exceeded.
    """
    path = [source]
    current_node = source

    while current_node != destination and len(path) < max_length:
        neighbors = list(G.successors(current_node))
        if not neighbors:
            return [] # Dead end

        # Choose a random neighbor that is not the previous node (to avoid immediate cycles)
        next_node = random.choice(neighbors)
        if len(path) > 1 and next_node == path[-2]:
            # Try another one, or just take the step if only one option
```

```
            pass

        path.append(next_node)
        current_node = next_node

    return path if current_node == destination else []


def levy_flight(path, G, destination):
    """
    Simulates a 'Lévy Flight' for path optimization.
    This creates a new "egg" (path) from an existing one, promoting global search.

    The discrete adaptation works by:
    1. Randomly selecting a segment of the current path (local search/mutation).
    2. Generating a new random path segment (Lévy jump) from the selected node
       to a point near the destination, then continuing to the destination.
    """
    if len(path) < 2: return generate_random_valid_path(G, path[0], destination)

    # 1. Select a random crossover point (excluding source and destination)
    if len(path) > 2:
        crossover_idx = random.randint(1, len(path) - 2)
    else:
        crossover_idx = 1 # Only one hop

    start_node = path[crossover_idx]

    # 2. Randomly decide to jump to a random neighbor or make a long jump
    if random.random() < 0.7: # Small step (Local search)
        new_segment = generate_random_valid_path(G, start_node, destination, max_length=5)
    else: # Large step (Global search, via a randomly chosen intermediate node)
        intermediate_nodes = [n for n in G.nodes() if n not in [start_node, destination]]
        if intermediate_nodes:
            intermediate_node = random.choice(intermediate_nodes)

            # Segment 1: Start node to intermediate node
            try:
                seg1 = nx.shortest_path(G, start_node, intermediate_node, weight='cost')
            except nx.NetworkXNoPath:
                seg1 = [start_node] # Fallback if no path
```

```
            # Segment 2: Intermediate node to destination
            try:
                seg2 = nx.shortest_path(G, intermediate_node, destination, weight='cost')
            except nx.NetworkXNoPath:
                seg2 = [destination]

            # Combine, removing duplicates
            new_segment = seg1 + seg2[1:]
        else:
            new_segment = generate_random_valid_path(G, start_node, destination, max_length=5)

    if not new_segment or new_segment[0] != start_node or new_segment[-1] != destination:
        # If new segment failed, try a totally new path from the start node of the segment
        new_segment = generate_random_valid_path(G, start_node, destination)

    # 3. Create the new solution: old path up to crossover, plus the new segment
    new_path = path[:crossover_idx] + new_segment

    # 4. Cleanup: Remove immediate cycles (e.g., A -> B -> A -> C)
    cleaned_path = []
    for node in new_path:
        if cleaned_path and node == cleaned_path[-1]: continue # Skip duplicates
        if len(cleaned_path) >= 2 and node == cleaned_path[-2]:
            cleaned_path.pop() # Remove the previous node to clear the cycle

        cleaned_path.append(node)

    return cleaned_path if cleaned_path[-1] == destination else []

# --- 3. Cuckoo Search Algorithm ---

def cuckoo_search_path_optimization(G, source, destination, N_nests=10, Pa=0.25, max_iter=100):

    print(f"\n--- Starting Cuckoo Search: {source} to {destination} ---\n")
    print(f"Parameters: Nests={N_nests}, Pa={Pa}, Max Iter={max_iter}\n")

    # Initialization: Generate an initial population of host nests (paths)
    nests = []
    for _ in range(N_nests):
        path = generate_random_valid_path(G, source, destination)
```

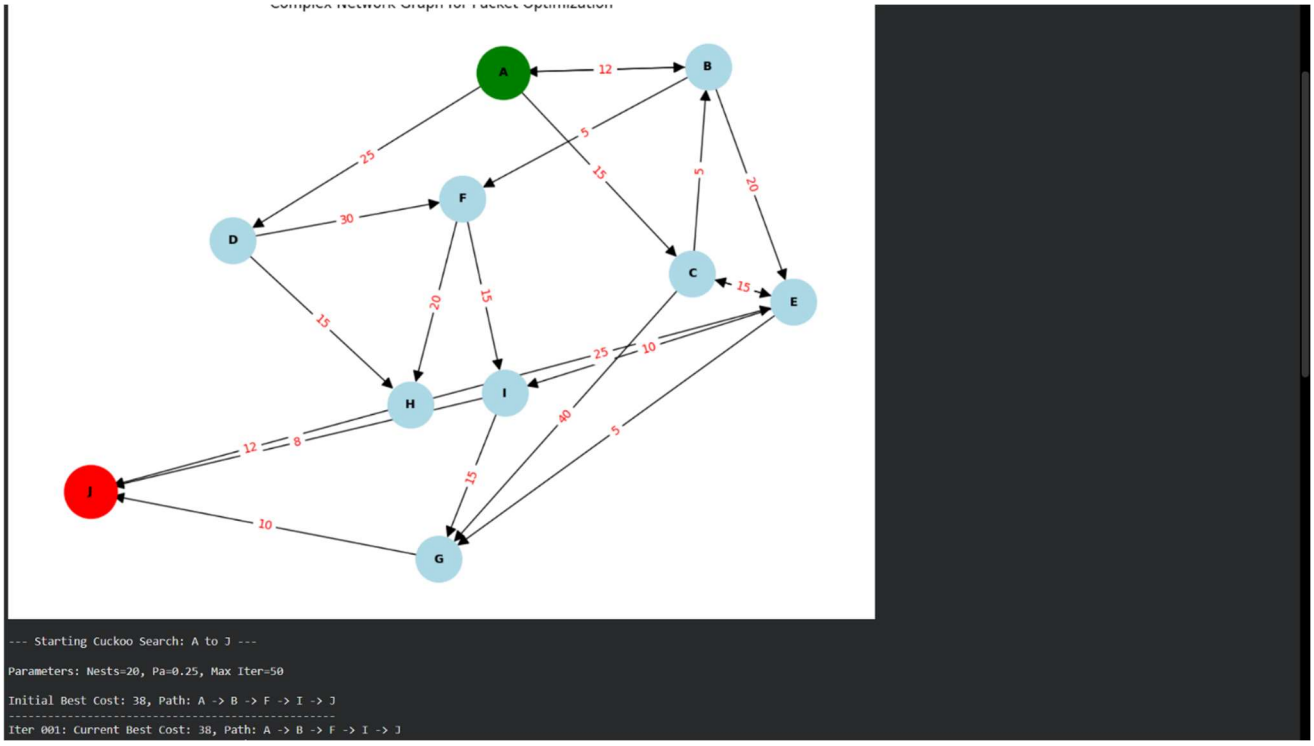
```
    if path:
        cost = calculate_path_cost(G, path)
        nests.append({'path': path, 'cost': cost})
    else:
        nests.append({'path': [], 'cost': float('inf')})

# Sort nests by cost (best is lowest cost)
nests.sort(key=lambda x: x['cost'])

best_path = nests[0]['path']
best_cost = nests[0]['cost']

if best_cost == float('inf'):
    print("Error: Could not find any initial valid path. Adjust graph or path generation.")
    return best_path, best_cost

print(f"Initial Best Cost: {best_cost}, Path: {' -> '.join(best_path)}")
print("-" * 50)

# Main optimization loop
for t in range(max_iter):
    # 1. Generate a new cuckoo egg (solution) via Lévy flight
    idx_cuckoo = random.randint(0, N_nests - 1)
    current_nest_path = nests[idx_cuckoo]['path']

    # Ensure we have a path to start the flight
    if not current_nest_path or current_nest_path[-1] != destination:
        current_nest_path = generate_random_valid_path(G, source, destination)
        if not current_nest_path: continue # Skip if path can't be generated

    new_cuckoo_path = levy_flight(current_nest_path, G, destination)
    new_cuckoo_cost = calculate_path_cost(G, new_cuckoo_path)

    # 2. Choose a random nest (j) to compare with
    idx_host = random.randint(0, N_nests - 1)
    host_cost = nests[idx_host]['cost']

    # 3. Replace nest (j) if the new cuckoo egg is better
    if new_cuckoo_cost < host_cost:
        nests[idx_host] = {'path': new_cuckoo_path, 'cost': new_cuckoo_cost}
```

```
        # 4. Abandon a fraction (Pa) of worse nests and build new ones
        num_abandon = int(N_nests * Pa)
        # The nests are already sorted, so we abandon the last 'num_abandon' ones (worst solutions)

        for i in range(1, num_abandon + 1):
            idx_worst = N_nests - i
            new_path = generate_random_valid_path(G, source, destination)
            if new_path:
                new_cost = calculate_path_cost(G, new_path)
                nests[idx_worst] = {'path': new_path, 'cost': new_cost}
            else:
                # In case new path generation fails, keep the original worst solution
                pass

        # 5. Re-sort the nests and update the global best
        nests.sort(key=lambda x: x['cost'])

        current_best_cost = nests[0]['cost']
        current_best_path = nests[0]['path']

        if current_best_cost < best_cost:
            best_cost = current_best_cost
            best_path = current_best_path

        # Print iteration results
        print(f"Iter {t+1:03d}: Current Best Cost: {current_best_cost}, Path: {' ->
'.join(current_best_path)}")


    print("-" * 50)
    return best_path, best_cost


# --- 4. Main Execution and Visualization ---

# 1. Create the complex graph
NETWORK_GRAPH = create_complex_graph()
SOURCE_NODE = 'A'
DESTINATION_NODE = 'J'

print("--- Network Graph Definition ---")
```

```
# 2. Print the graph details
print("Nodes:", list(NETWORK_GRAPH.nodes()))
print("Edges and Costs (Sample):")
for u, v, data in list(NETWORK_GRAPH.edges(data=True))[:5]:
   print(f"  {u} -> {v}: Cost {data['cost']}")
print(f"Total Edges: {NETWORK_GRAPH.number_of_edges()}")

# 3. Visualize the graph
plt.figure(figsize=(10, 7))
pos = nx.spring_layout(NETWORK_GRAPH, seed=42) # Layout for visualization
nx.draw(NETWORK_GRAPH, pos, with_labels=True, node_size=1500, node_color='lightblue',
        font_size=10, font_weight='bold', arrowsize=20)

# Draw edge labels (costs)
edge_labels = nx.get_edge_attributes(NETWORK_GRAPH, 'cost')
nx.draw_networkx_edge_labels(NETWORK_GRAPH, pos, edge_labels=edge_labels,
font_color='red')

# Highlight source and destination
nx.draw_networkx_nodes(NETWORK_GRAPH, pos, nodelist=[SOURCE_NODE],
node_color='green', node_size=2000, label='Source')
nx.draw_networkx_nodes(NETWORK_GRAPH, pos, nodelist=[DESTINATION_NODE],
node_color='red', node_size=2000, label='Destination')

plt.title("Complex Network Graph for Packet Optimization")
#
plt.show()

# 4. Run the Cuckoo Search optimization
FINAL_BEST_PATH, FINAL_BEST_COST = cuckoo_search_path_optimization(
   G=NETWORK_GRAPH,
   source=SOURCE_NODE,
   destination=DESTINATION_NODE,
   N_nests=20, # Number of solutions to maintain
   Pa=0.25,    # Probability of abandoning the worst nest
   max_iter=50  # Number of generations
)

# 5. Print the final best result
print("\n" * 2)
```

```
print("=" * 60)
print(f"FINAL BEST PATH FOUND BY CUCKOO SEARCH")
print(f"   BEST PATH: {' -> '.join(FINAL_BEST_PATH)}")
print(f"   BEST COST: {FINAL_BEST_COST}")
print("=" * 60)
```

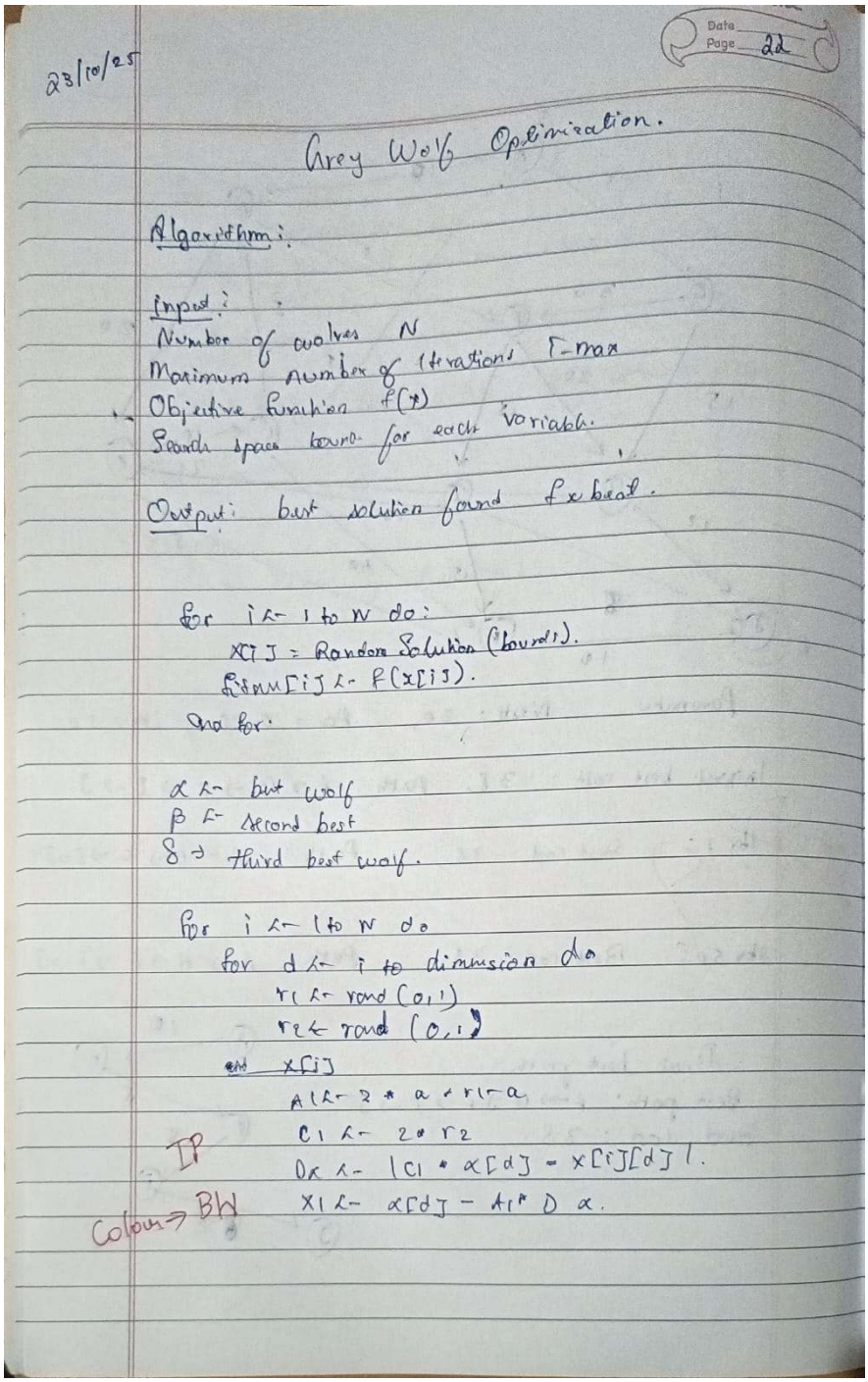
Output:



Program 6:

Grey Wolf Optimization for Image Preocessing Color to black and white

Algorithm:



r1 ← rand (0,1)
r2 ← rand (0,1)
A2 ← 2 * a * r1 - a
C2 ← 2 * r2
Dβ ← |C2 * β[d] - x[i][d]|
x2 ← β[d] - A2 * Dβ

r1 ← rand (0,1)
r2 ← rand (0,1)
A3 ← 2 * a * r1 - a
C3 ← r1 r2
Dδ ← |C3 * δ[d] - x[i][d]|
x3 ← δ[d] - A3 * Dδ

x_new[d] ← (x1 + x2 + x3)/3
end for.

x[i] ← Apply Bounds (x_new, bounds)
fitness[i] ← f(x[i])
end for.

Update α, β, δ based on new fitness values
end for

return α, f(α).

Code:

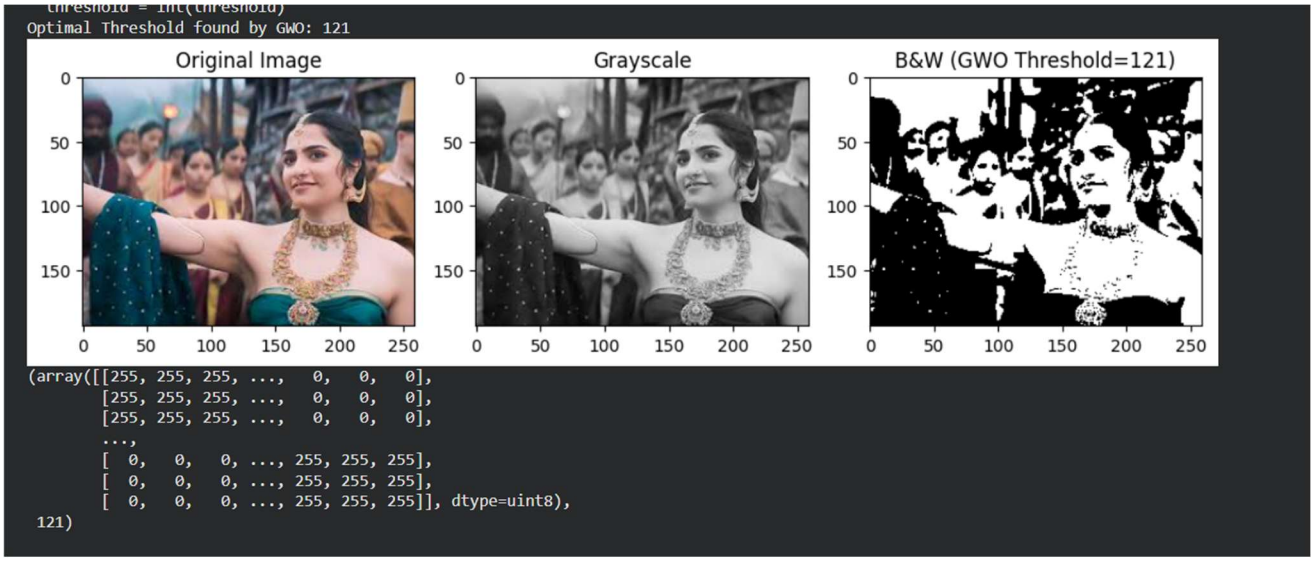
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# ----------------------------
# Grey Wolf Optimization (GWO)
# ----------------------------
def gwo_optimize(obj_func, lb, ub, dim, num_agents=10, max_iter=50):
    # Initialize the positions of search agents
    positions = np.random.uniform(lb, ub, (num_agents, dim))

    # Initialize alpha, beta, and delta wolves
    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)
    alpha_score = float('inf')
    beta_score = float('inf')
    delta_score = float('inf')

    # Main loop
    for t in range(max_iter):
        for i in range(num_agents):
            # Keep search agents within bounds
            positions[i] = np.clip(positions[i], lb, ub)

            # Calculate fitness
            fitness = obj_func(positions[i])

            # Update alpha, beta, delta
            if fitness < alpha_score:
                delta_score = beta_score
                delta_pos = beta_pos.copy()
                beta_score = alpha_score
                beta_pos = alpha_pos.copy()
                alpha_score = fitness
                alpha_pos = positions[i].copy()
            elif fitness < beta_score:
                delta_score = beta_score
                delta_pos = beta_pos.copy()
```

```
            beta_score = fitness
            beta_pos = positions[i].copy()
        elif fitness < delta_score:
            delta_score = fitness
            delta_pos = positions[i].copy()

    # Coefficients a, A, and C
    a = 2 - t * (2 / max_iter)

    # Update positions
    for i in range(num_agents):
        for j in range(dim):
            r1, r2 = np.random.rand(), np.random.rand()
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_pos[j] - positions[i][j])
            X1 = alpha_pos[j] - A1 * D_alpha

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

            r1, r2 = np.random.rand(), np.random.rand()
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_pos[j] - positions[i][j])
            X3 = delta_pos[j] - A3 * D_delta

            positions[i][j] = (X1 + X2 + X3) / 3.0

    return alpha_pos, alpha_score


# ----------------------------
# Objective Function (Otsu's variance)
# ----------------------------
def otsu_threshold_objective(threshold, gray_img):
    threshold = int(threshold)
    if threshold <= 0 or threshold >= 255:
```

```
        return float('inf')

    hist, _ = np.histogram(gray_img, bins=256, range=(0, 256))
    hist = hist.astype(np.float32)
    total = gray_img.size

    w0 = np.sum(hist[:threshold]) / total
    w1 = np.sum(hist[threshold:]) / total
    if w0 == 0 or w1 == 0:
        return float('inf')

    m0 = np.sum(np.arange(threshold) * hist[:threshold]) / (np.sum(hist[:threshold]) + 1e-6)
    m1 = np.sum(np.arange(threshold, 256) * hist[threshold:]) / (np.sum(hist[threshold:]) + 1e-6)

    # Between-class variance (maximize, so we return negative for minimization)
    var_between = w0 * w1 * (m0 - m1) ** 2
    return -var_between


# ----------------------------
# Main function
# ----------------------------
def gwo_image_threshold(image_path):
    # Read and convert to grayscale
    img = cv2.imread(image_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Define objective wrapper
    def obj_func(threshold):
        return otsu_threshold_objective(threshold, gray)

    # Run GWO
    best_threshold, best_score = gwo_optimize(obj_func, lb=0, ub=255, dim=1, num_agents=15,
max_iter=50)
    best_threshold = int(best_threshold[0])
    print(f"Optimal Threshold found by GWO: {best_threshold}")

    # Apply threshold
    _, bw_img = cv2.threshold(gray, best_threshold, 255, cv2.THRESH_BINARY)

    # Display results
```

```
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))

    plt.subplot(1, 3, 2)
    plt.title("Grayscale")
    plt.imshow(gray, cmap='gray')

    plt.subplot(1, 3, 3)
    plt.title(f"B&W (GWO Threshold={best_threshold})")
    plt.imshow(bw_img, cmap='gray')
    plt.tight_layout()
    plt.show()

    return bw_img, best_threshold

gwo_image_threshold("download.jpeg")
```

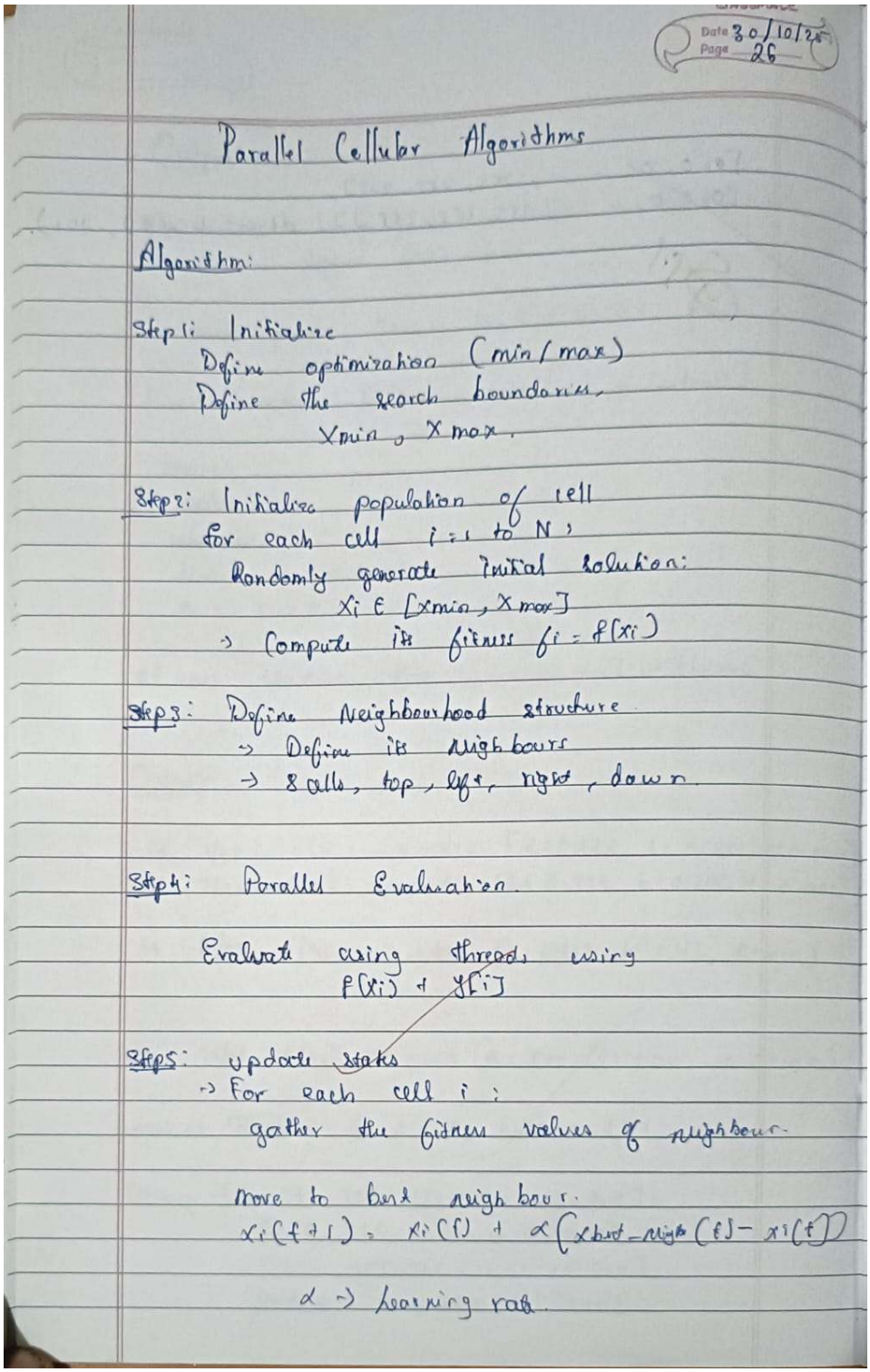
Output:



Program 7:

Parallel Cellular Algorithm for Routing

Algorithm:



Parallel Cellular Algorithms

Algorithm:

Step 1: Initialize
Define optimization (min / max)
Define the search boundaries.
Xmin, Xmax.

Step 2: Initialize population of cell
for each cell i = 1 to N,
Randomly generate initial solution:
$x_i \in [x_{min}, x_{max}]$
→ Compute its fitness $f_i = f(x_i)$

Step 3: Define Neighbourhood structure
→ Define its neighbours
→ 8 cells, top, left, right, down.

Step 4: Parallel Evaluation.

Evaluate using threads, using
f(xi) & yfij

Step 5: update states
→ For each cell i:
gather the fitness values of neighbour.

move to best neighbour.
$x_i(t+1) = x_i(t) + \alpha(x_{best\_neigh}(t) - x_i(t))$

α → learning rate.

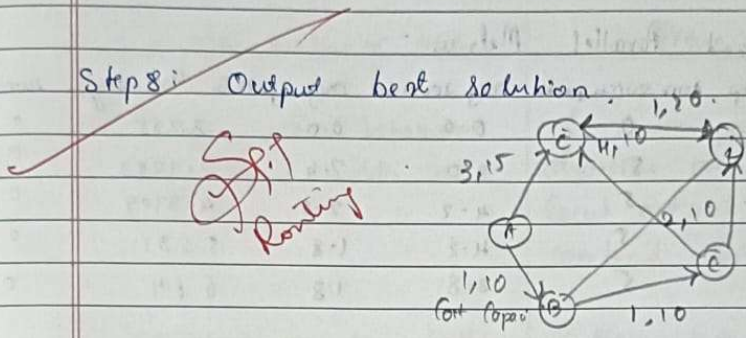
Step6: New fitness

$f_i(t+1) = f(x_i(t+1))$

$f_i(t+1) < f_{best}$ update x_{best}.

Step7: Repeat until convergence.

Step8: Output best solution.



Output:

Average Successful Router per step (Parallel): 4.16
Average Successful Router per step (Sequential): 3.98
Average Traffic load at End (parallel): 12.51
Average Traffic load at End (Sequential): 12.57

Starting optimized parallel cellular Simulation with Routing...
Optimized Parallel Time Step 10/100 completed.
Optimized Parallel Time Step 20/100 completed.
.
.
Optimized Parallel Time Step 100/100 completed.

Code:

```
import networkx as nx
import random
import time
import pandas as pd
import heapq
from concurrent.futures import ProcessPoolExecutor, as_completed
import multiprocessing

# 1. Define the network structure
G = nx.DiGraph()
edges = [
    ('A', 'B', {'cost': 1, 'capacity': 10}),
    ('A', 'C', {'cost': 3, 'capacity': 15}),
    ('B', 'C', {'cost': 1, 'capacity': 10}),
    ('B', 'D', {'cost': 2, 'capacity': 10}),
    ('C', 'D', {'cost': 1, 'capacity': 15}),
    ('C', 'E', {'cost': 4, 'capacity': 10}),
    ('D', 'E', {'cost': 1, 'capacity': 20})
]
G.add_edges_from(edges)

# Define source-destination pairs for routing attempts
source_destination_pairs = [
    ('A', 'D'),
    ('A', 'E'),
    ('B', 'E'),
    ('C', 'D')
]

# Initialize CA grid state and simulation parameters
initial_pheromone = 0.1
ca_grid_state_initial = {}
for u, v in G.edges():
    ca_grid_state_initial[(u, v)] = {
        'is_active_route': 0,
        'pheromone_level': initial_pheromone,
        'traffic_load': 0
```

```
    }

simulation_parameters = {
    'num_time_steps': 100,
    'pheromone_evaporation_rate': 0.05,
    'pheromone_deposit_amount': 0.1,
    'random_exploration_factor': 0.01,
    'traffic_increase_factor': 0.01,
    'traffic_decrease_factor': 0.02,
    'route_deactivation_threshold': 0.8,
    'pheromone_deposit_threshold': 0.2,
    'activation_factor': 0.05,
    'routing_attempts_per_step': 5,
    'pheromone_deposit_on_success': 0.5,
    'dijkstra_traffic_factor': 0.5,
    'dijkstra_pheromone_factor': 0.05
}

# Helper function to get neighborhood (from previous step)
def get_neighborhood(u, v, graph):
    neighborhood = []
    for in_edge_u, in_edge_v in graph.in_edges(u):
        neighborhood.append((in_edge_u, in_edge_v))
    for out_edge_u, out_edge_v in graph.out_edges(v):
        neighborhood.append((out_edge_u, out_edge_v))
    return neighborhood

# Helper function to update cell state (from previous step)
def update_cell_state(u, v, current_state, neighbor_states, graph, simulation_parameters):
    new_state = current_state.copy()
    pheromone_evaporation_rate = simulation_parameters['pheromone_evaporation_rate']
    pheromone_deposit_amount = simulation_parameters['pheromone_deposit_amount']
    random_exploration_factor = simulation_parameters['random_exploration_factor']
    traffic_increase_factor = simulation_parameters['traffic_increase_factor']
    traffic_decrease_factor = simulation_parameters['traffic_decrease_factor']
    route_deactivation_threshold = simulation_parameters['route_deactivation_threshold']

    new_state['pheromone_level'] *= (1 - pheromone_evaporation_rate)

    for (nu, nv), state in neighbor_states.items():
        if state['pheromone_level'] > simulation_parameters['pheromone_deposit_threshold']:
```

```
            new_state['pheromone_level'] += pheromone_deposit_amount * state['pheromone_level']

    max_pheromone = simulation_parameters.get('max_pheromone', float('inf'))
    new_state['pheromone_level'] = min(new_state['pheromone_level'], max_pheromone)

    active_neighbors = sum(state['is_active_route'] for state in neighbor_states.values())
    new_state['traffic_load'] += active_neighbors * traffic_increase_factor
    new_state['traffic_load'] *= (1 - traffic_decrease_factor)

    edge_capacity = graph[u][v].get('capacity', float('inf'))
    new_state['traffic_load'] = min(new_state['traffic_load'], edge_capacity)
    graph[u][v]['traffic_load'] = new_state['traffic_load']

    cost = graph[u][v].get('cost', 1)
    available_capacity = edge_capacity - new_state['traffic_load']
    attractiveness = (new_state['pheromone_level'] + 1e-6) / (cost + 1e-6) * (available_capacity + 1e-
6)

    activation_probability = attractiveness * simulation_parameters['activation_factor']

    if random.random() < random_exploration_factor:
        activation_probability = random.random()

    if new_state['is_active_route'] == 0:
        if random.random() < activation_probability:
            new_state['is_active_route'] = 1
    else:
        if new_state['traffic_load'] > edge_capacity * route_deactivation_threshold:
            new_state['is_active_route'] = 0
        elif random.random() > activation_probability and random.random() > (1 -
random_exploration_factor):
            new_state['is_active_route'] = 0

    return new_state

# Pathfinding function using Dijkstra's algorithm (optimized version)
def find_route_with_dijkstra_ca(source, destination, graph, ca_grid_state):
    def get_dynamic_weight(u, v):
        edge_state = ca_grid_state.get((u, v), {'is_active_route': 0, 'pheromone_level': 0, 'traffic_load':
float('inf')})
        original_cost = graph[u][v].get('cost', 1)
```

```
        traffic_load = edge_state.get('traffic_load', 0)
        pheromone_level = edge_state.get('pheromone_level', 0)

        alpha = simulation_parameters.get('dijkstra_traffic_factor', 0.1)
        beta = simulation_parameters.get('dijkstra_pheromone_factor', 0.01)

        dynamic_weight = original_cost + alpha * traffic_load - beta * pheromone_level
        return max(0, dynamic_weight)

    distances = {node: float('inf') for node in graph.nodes()}
    distances[source] = 0
    previous_nodes = {}
    priority_queue = [(0, source)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_node == destination:
            path = []
            while current_node is not None:
                path.append(current_node)
                current_node = previous_nodes.get(current_node)
            return path[::-1]

        if current_distance > distances[current_node]:
            continue

        for neighbor in graph.neighbors(current_node):
            edge = (current_node, neighbor)
            if edge in graph.edges():
                weight = get_dynamic_weight(current_node, neighbor)
                distance = current_distance + weight

                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    previous_nodes[neighbor] = current_node
                    heapq.heappush(priority_queue, (distance, neighbor))

    return None

# Parallel update function (from previous step)
```

```
def update_cell_state_parallel(edge_chunk, ca_grid_state, graph, simulation_parameters):
    updated_chunk_state = {}
    for u, v in edge_chunk:
        current_state = ca_grid_state[(u, v)]
        neighborhood = get_neighborhood(u, v, graph)
        neighbor_states = {
            (nu, nv): ca_grid_state.get((nu, nv), {'is_active_route': 0, 'pheromone_level': 0, 'traffic_load':
0})
            for (nu, nv) in neighborhood
        }
        new_state = update_cell_state(u, v, current_state, neighbor_states, graph,
simulation_parameters)
        updated_chunk_state[(u, v)] = new_state
    return updated_chunk_state

# Metrics calculation function (from previous step)
def calculate_metrics(ca_grid_history, graph, simulation_parameters, successful_routes_history):
    num_time_steps = simulation_parameters['num_time_steps']
    metrics = {
        'time_step': list(range(num_time_steps + 1)),
        'num_successful_routes': [0],
        'avg_route_cost': [0],
        'avg_path_length': [0],
        'avg_traffic_load': [],
        'num_overloaded_edges': []
    }

    for t in range(num_time_steps + 1):
        current_ca_state = ca_grid_history[t]

        total_traffic_load = 0
        overloaded_edges_count = 0
        num_edges = 0
        for (u, v), state in current_ca_state.items():
            num_edges += 1
            traffic_load = state.get('traffic_load', 0)
            capacity = graph[u][v].get('capacity', float('inf'))

            total_traffic_load += traffic_load
            if capacity != float('inf') and traffic_load > capacity:
                overloaded_edges_count += 1
```

```
        metrics['avg_traffic_load'].append(total_traffic_load / num_edges if num_edges > 0 else 0)
        metrics['num_overloaded_edges'].append(overloaded_edges_count)

        if t > 0:
            successful_routes_this_step = successful_routes_history[t-1]
            metrics['num_successful_routes'].append(len(successful_routes_this_step))

            if successful_routes_this_step:
                total_cost_this_step = 0
                total_length_this_step = 0
                for source, destination, route in successful_routes_this_step:
                    route_cost = sum(graph[route[i]][route[i+1]].get('cost', 1) for i in range(len(route) - 1))
                    total_cost_this_step += route_cost
                    total_length_this_step += len(route) - 1

                metrics['avg_route_cost'].append(total_cost_this_step / len(successful_routes_this_step))
                metrics['avg_path_length'].append(total_length_this_step /
len(successful_routes_this_step))
            else:
                metrics['avg_route_cost'].append(0)
                metrics['avg_path_length'].append(0)

    return metrics

# Main simulation runner (adapted from previous step)
def run_simulation_optimized_routing(graph, initial_ca_grid_state, simulation_parameters,
source_destination_pairs, parallel=True):
    if parallel:
        print("\nStarting Optimized Parallel Cellular Automaton Simulation with Routing...")
    else:
        print("\nStarting Optimized Sequential Cellular Automaton Simulation with Routing...")

    ca_grid_state_run = initial_ca_grid_state.copy()
    ca_grid_history_run = [ca_grid_state_run.copy()]
    successful_routes_history_run = []

    start_time = time.time()

    for t in range(simulation_parameters['num_time_steps']):
        for edge in graph.edges():
```

```
            ca_grid_state_run[edge]['is_active_route'] = 0

        successful_routes_this_step = []
        for _ in range(simulation_parameters['routing_attempts_per_step']):
            source, destination = random.choice(source_destination_pairs)
            found_route = find_route_with_dijkstra_ca(source, destination, graph, ca_grid_state_run)

            if found_route:
                successful_routes_this_step.append((source, destination, found_route))
                for i in range(len(found_route) - 1):
                    u, v = found_route[i], found_route[i+1]
                    edge = (u, v)
                    if edge in ca_grid_state_run:
                        ca_grid_state_run[edge]['pheromone_level'] +=
simulation_parameters['pheromone_deposit_on_success']
                        ca_grid_state_run[edge]['is_active_route'] = 1
                        ca_grid_state_run[edge]['traffic_load'] += 1
                        edge_capacity = graph[u][v].get('capacity', float('inf'))
                        ca_grid_state_run[edge]['traffic_load'] = min(ca_grid_state_run[edge]['traffic_load'],
edge_capacity)
                        graph[u][v]['traffic_load'] = ca_grid_state_run[edge]['traffic_load']

        successful_routes_history_run.append(successful_routes_this_step)

        next_ca_grid_state_run = {}
        edges_list = list(graph.edges())

        if parallel:
            num_processes = multiprocessing.cpu_count()
            chunk_size = max(1, len(edges_list) // num_processes)
            edge_chunks = [edges_list[i:i + chunk_size] for i in range(0, len(edges_list), chunk_size)]

            with ProcessPoolExecutor(max_workers=num_processes) as executor:
                future_to_chunk = {
                    executor.submit(update_cell_state_parallel, chunk, ca_grid_state_run.copy(), graph,
simulation_parameters): chunk
                    for chunk in edge_chunks
                }

                for future in as_completed(future_to_chunk):
                    try:
```

```
                    updated_chunk_state = future.result()
                    next_ca_grid_state_run.update(updated_chunk_state)
                except Exception as exc:
                    print(f'Edge chunk generated an exception: {exc}')
        else:
            for u, v in graph.edges():
                current_state = ca_grid_state_run[(u, v)]
                neighborhood = get_neighborhood(u, v, graph)
                neighbor_states = {
                    (nu, nv): ca_grid_state_run.get((nu, nv), {'is_active_route': 0, 'pheromone_level': 0,
'traffic_load': 0})
                    for (nu, nv) in neighborhood
                }
                new_state = update_cell_state(u, v, current_state, neighbor_states, graph,
simulation_parameters)
                next_ca_grid_state_run[(u, v)] = new_state

        ca_grid_state_run = next_ca_grid_state_run
        ca_grid_history_run.append(ca_grid_state_run.copy())

        if (t + 1) % 10 == 0:
            if parallel:
                print(f"Optimized Parallel Time Step {t + 1}/{simulation_parameters['num_time_steps']}
completed.")
            else:
                print(f"Optimized Sequential Time Step {t +
1}/{simulation_parameters['num_time_steps']} completed.")

    end_time = time.time()
    execution_time = end_time - start_time
    if parallel:
        print(f"Optimized Parallel Simulation Finished in {execution_time:.4f} seconds.")
    else:
        print(f"Optimized Sequential Simulation Finished in {execution_time:.4f} seconds.")

    return ca_grid_history_run, successful_routes_history_run, execution_time

# --- Run optimized simulations and compare ---
# Reset graph traffic load before running simulations
for u,v in G.edges():
    G[u][v]['traffic_load'] = 0
```

```
# Run optimized parallel simulation
ca_grid_history_opt_parallel, successful_routes_history_opt_parallel, execution_time_opt_parallel =
run_simulation_optimized_routing(
    G.copy(),
    ca_grid_state_initial.copy(),
    simulation_parameters,
    source_destination_pairs,
    parallel=True
)

# Reset graph traffic load for sequential run
for u,v in G.edges():
    G[u][v]['traffic_load'] = 0

# Run optimized sequential simulation
ca_grid_history_opt_sequential, successful_routes_history_opt_sequential,
execution_time_opt_sequential = run_simulation_optimized_routing(
    G.copy(),
    ca_grid_state_initial.copy(),
    simulation_parameters,
    source_destination_pairs,
    parallel=False
)

# Calculate metrics for optimized simulations
print("\nCalculating Metrics for Optimized Simulations...")
parallel_metrics_opt = calculate_metrics(ca_grid_history_opt_parallel, G, simulation_parameters,
successful_routes_history_opt_parallel)
sequential_metrics_opt = calculate_metrics(ca_grid_history_opt_sequential, G,
simulation_parameters, successful_routes_history_opt_sequential)

parallel_metrics_opt_df = pd.DataFrame(parallel_metrics_opt)
sequential_metrics_opt_df = pd.DataFrame(sequential_metrics_opt)

print("\nOptimized Parallel Metrics:")
display(parallel_metrics_opt_df.head())

print("\nOptimized Sequential Metrics:")
display(sequential_metrics_opt_df.head())
```

```
print(f"\nOptimized Parallel Execution Time: {execution_time_opt_parallel:.4f} seconds")
print(f'Optimized Sequential Execution Time: {execution_time_opt_sequential:.4f} seconds")

# Compare average successful routes
avg_successful_routes_opt_parallel = parallel_metrics_opt_df['num_successful_routes'].mean()
avg_successful_routes_opt_sequential = sequential_metrics_opt_df['num_successful_routes'].mean()

print(f"\nAverage Successful Routes per Step (Optimized Parallel):
{avg_successful_routes_opt_parallel:.2f}")
print(f'Average Successful Routes per Step (Optimized Sequential):
{avg_successful_routes_opt_sequential:.2f}")

# Compare average traffic load at the end
avg_traffic_end_opt_parallel = parallel_metrics_opt_df['avg_traffic_load'].iloc[-1]
avg_traffic_end_opt_sequential = sequential_metrics_opt_df['avg_traffic_load'].iloc[-1]

print(f"\nAverage Traffic Load at End (Optimized Parallel): {avg_traffic_end_opt_parallel:.2f}")
print(f'Average Traffic Load at End (Optimized Sequential): {avg_traffic_end_opt_sequential:.2f}")
```

Output:

```
Sequential Time Step 70/100 completed.
Sequential Time Step 80/100 completed.
Sequential Time Step 90/100 completed.
Sequential Time Step 100/100 completed.
Sequential Simulation Finished in 0.0219 seconds.

Calculating Metrics...

Parallel Metrics:
```

	time_step	num_successful_routes	avg_route_cost	avg_path_length	avg_traffic_load	num_overloaded_edges
0	0	0	0.00	0.0	1.857143	0
1	1	5	4.00	2.6	2.985257	0
2	2	4	2.75	2.0	4.366723	0
3	3	4	2.75	2.5	6.010475	0
4	4	5	2.60	2.4	7.335637	0

```
Sequential Metrics:
```

	time_step	num_successful_routes	avg_route_cost	avg_path_length	avg_traffic_load	num_overloaded_edges
0	0	0	0.000000	0.0	1.428571	0
1	1	5	4.200000	2.0	3.136686	0
2	2	4	4.250000	3.0	4.950695	0
3	3	5	4.800000	2.6	5.727024	0
4	4	3	4.333333	2.0	7.052255	0

```
Parallel Execution Time: 3.6789 seconds
Sequential Execution Time: 0.0219 seconds

Average Successful Routes per Step (Parallel): 4.16
Average Successful Routes per Step (Sequential): 3.98

Average Traffic Load at End (Parallel): 12.51
Average Traffic Load at End (Sequential): 12.57

Number of Overloaded Edges at End (Parallel): 0
Number of Overloaded Edges at End (Sequential): 0
```