**Name: K.Prajiith**

**Reg.no:192324013**

**Problem 1: Optimizing Delivery Routes (Case Study).**

**Scenario:** You are working for a logistics company that wants to optimize its delivery routes to minimize fuel consumption and delivery time. The company operates in a city with a complex road network.

**Tasks:**

1. Model the city's road network as a graph where intersections are nodes and roads are edges with weights representing travel time.

2. Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations.

3. Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.

**Deliverables:**

● Graph model of the city's road network.

● Pseudocode and implementation of Dijkstra’s algorithm.

● Analysis of the algorithm’s efficiency and potential improvements.

**Reasoning:** Explain why Dijkstra’s algorithm is suitable for this problem. Discuss any assumptions made (e.g., non-negative weights) and how different road conditions (e.g., traffic, road closures) could affect your solution.

**Code:**

graph = {

'A': {'B': 4, 'C': 2},

'B': {'A': 4, 'C': 1, 'D': 5},

'C': {'A': 2, 'B': 1, 'D': 8, 'E': 10},

'D': {'B': 5, 'C': 8, 'E': 2, 'F': 6},

'E': {'C': 10, 'D': 2, 'F': 2},

'F': {'D': 6, 'E': 2}

}

start\_node = 'A'

distances = {}

predecessors = {}

for node in graph:

distances[node] = float('inf')

predecessors[node] = None

distances[start\_node] = 0

priority\_queue = [(0, start\_node)]

visited = set()

while priority\_queue:

priority\_queue.sort()

current\_distance, current\_node = priority\_queue.pop(0)

if current\_node in visited:

continue

visited.add(current\_node)

for neighbor, weight in graph[current\_node].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

predecessors[neighbor] = current\_node

priority\_queue.append((distance, neighbor))

print("Shortest distances from the start node:")

for node in distances:

print(f"Distance to {node}: {distances[node]}")

print("\nShortest paths from the start node:")

for node in graph:

if node == start\_node:

continue

path = []

current = node

while current is not None:

path.append(current)

current = predecessors[current]

path.reverse()

print(f"Path to {node}: {' -> '.join(path)}")

TIME COMPLEXITY: nlog(n)

Space complexity:n

**Problem 2:** **Dynamic Pricing Algorithm for E-commerce**

**Scenario:** An e-commerce company wants to implement a dynamic pricing algorithm to adjust the prices of products in real-time based on demand and competitor prices.

**Tasks:**

1. Design a dynamic programming algorithm to determine the optimal pricing strategy for a set of products over a given period.

2. Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm.

3. Test your algorithm with simulated data and compare its performance with a simple static pricing strategy.

**Deliverables:**

● Pseudocode and implementation of the dynamic pricing algorithm.

● Simulation results comparing dynamic and static pricing strategies.

● Analysis of the benefits and drawbacks of dynamic pricing.

**Reasoning:** Justify the use of dynamic programming for this problem. Explain how you incorporated different factors into your algorithm and discuss any challenges faced during implementation.

**Code:**

def dynamic\_pricing(T, P, I, E, C):

n = len(P)

dp = [[0 for \_ in range(n)] for \_ in range(T)]

ps = [[0 for \_ in range(n)] for \_ in range(T)]

for t in range(T):

for i in range(n):

mp = float('-inf')

op = 0

for price in range(C[i] - 10, C[i] + 11):

if I[i] <= 0:

break

demand = max(E[i] \* (price - C[i]), 0)

profit = price \* demand

if t > 0:

profit += dp[t-1][i]

if profit > mp:

mp = profit

op = price

dp[t][i] = mp

ps[t][i] = op

I[i] -= max(E[i] \* (op - C[i]), 0)

return ps, dp[T-1]

T = 4

P = ['P1', 'P2','P3','P4']

I = [100, 100,150,200]

E = [1.2, 1.1,0.8,0.9]

C = [50, 60,70,80]

opt, finprofit = dynamic\_pricing(T, P, I, E, C)

print("Optimal Prices (Dynamic): ", opt)

print("Final Profits (Dynamic): ", finprofit)

Time complexity: O(n)

Sace complexity: O(n)

**Problem 3: Social Network Analysis (Case Study)**

**Scenario:** A social media company wants to identify influential users within its network to target for marketing campaigns.

**Tasks:**

1. Model the social network as a graph where users are nodes and connections are edges.

2. Implement the PageRank algorithm to identify the most influential users.

3. Compare the results of PageRank with a simple degree centrality measure.

**Deliverables:**

● Graph model of the social network.

● Pseudocode and implementation of the PageRank algorithm.

● Comparison of PageRank and degree centrality results.

**Reasoning:** Discuss why PageRank is an effective measure for identifying influential users. Explain the differences between PageRank and degree centrality and why one might be preferred over the other in different scenarios.

**Code:**

class Graph:

def \_init\_(self):

self.graph = {}

def add\_edge(self, u, v):

if u not in self.graph:

self.graph[u] = []

self.graph[u].append(v)

def pagerank(self, damping\_factor=0.85, max\_iterations=100, convergence\_threshold=0.0001):

nodes = list(self.graph.keys())

N = len(nodes)

pr = {node: 1 / N for node in nodes}

out\_degree = {node: len(self.graph[node]) for node in nodes}

def incoming\_neighbors(node):

neighbors = []

for u in self.graph:

if node in self.graph[u]:

neighbors.append(u)

return neighbors

def converge(new\_pr, old\_pr, threshold):

for node in new\_pr:

if abs(new\_pr[node] - old\_pr[node]) >= threshold:

return False

return True

for \_ in range(max\_iterations):

new\_pr = {}

for node in nodes:

new\_rank = (1 - damping\_factor) / N

for neighbor in incoming\_neighbors(node):

new\_rank += damping\_factor \* pr[neighbor] / out\_degree[neighbor]

new\_pr[node] = new\_rank

if converge(new\_pr, pr, convergence\_threshold):

break

pr = new\_pr

return pr

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

g.add\_edge('D', 'A')

g.add\_edge('D', 'C')

pagerank\_scores = g.pagerank()

print("PageRank scores:", pagerank\_scores)

**Time Complexity: O(max\_iterations×∣V∣2)**

**Space Complexity: O(∣V∣+∣E∣)**

**Problem 4: Fraud Detection in Financial Transactions**

**Scenario:** A financial institution wants to develop an algorithm to detect fraudulent transactions in real-time.

**Tasks:**

1. Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules (e.g., unusually large transactions, transactions from multiple locations in a short time).

2. Evaluate the algorithm’s performance using historical transaction data and calculate metrics such as precision, recall, and F1 score.

3. Suggest and implement potential improvements to the algorithm.

**Deliverables:**

● Pseudocode and implementation of the fraud detection algorithm.

● Performance evaluation using historical data.

● Suggestions and implementation of improvements.

**Reasoning:** Explain why a greedy algorithm is suitable for real-time fraud detection. Discuss the trade-offs between speed and accuracy and how your algorithm addresses them.

**Code:**

class Transaction:

def \_init\_(self, amount, locations):

self.amount = amount

self.locations = locations

def detect\_fraud(transactions, threshold\_amount=1000, threshold\_count=3, short\_time\_period=3600):

flagged\_transactions = []

for transaction in transactions:

if is\_fraudulent(transaction, threshold\_amount, threshold\_count, short\_time\_period):

flagged\_transactions.append(transaction)

return flagged\_transactions

def is\_fraudulent(transaction, threshold\_amount, threshold\_count, short\_time\_period):

if transaction.amount > threshold\_amount:

return True

if len(transaction.locations) > threshold\_count and are\_locations\_close(transaction.locations, short\_time\_period):

return True

return False

def are\_locations\_close(locations, short\_time\_period):

# Here should be the logic to check if transactions occurred within a short time period.

# This function should return True if the locations are close in terms of transaction times.

# For the sake of simplicity, we'll assume all locations are close.

return True

# Example transactions

transactions = [

Transaction(amount=1500, locations=['LocationA', 'LocationB', 'LocationC']),

Transaction(amount=500, locations=['LocationA']),

Transaction(amount=1200, locations=['LocationB']),

Transaction(amount=2000, locations=['LocationC']),

Transaction(amount=800, locations=['LocationD']),

Transaction(amount=300, locations=['LocationA']),

Transaction(amount=1100, locations=['LocationE']),

]

flagged\_transactions = detect\_fraud(transactions, threshold\_amount=1000, threshold\_count=2, short\_time\_period=3600)

print("Flagged Transactions:")

for transaction in flagged\_transactions:

print(f"Amount: {transaction.amount}, Locations: {transaction.locations}")

# Actual true labels for the transactions (1 for fraudulent, 0 for non-fraudulent)

true\_labels = [1, 0, 1, 1, 0, 0, 1]

# Predicted labels based on the detection algorithm (1 for flagged, 0 for not flagged)

predicted\_labels = [1 if transaction in flagged\_transactions else 0 for transaction in transactions]

def precision\_recall\_f1(true\_labels, predicted\_labels):

tp = sum(1 for t, p in zip(true\_labels, predicted\_labels) if t == 1 and p == 1)

fp = sum(1 for t, p in zip(true\_labels, predicted\_labels) if t == 0 and p == 1)

fn = sum(1 for t, p in zip(true\_labels, predicted\_labels) if t == 1 and p == 0)

precision = tp / (tp + fp) if (tp + fp) > 0 else 0

recall = tp / (tp + fn) if (tp + fn) > 0 else 0

f1 = 2 \* precision \* recall / (precision + recall) if (precision + recall) > 0 else 0

return precision, recall, f1

precision, recall, f1 = precision\_recall\_f1(true\_labels, predicted\_labels)

print(f"Precision: {precision:.2f}")

print(f"Recall: {recall:.2f}")

print(f"F1 Score: {f1:.2f}")

TIME COMPLEXITY:O(n)

SPACE COMPLEXITY:O(n)

**Problem 5: Real-Time Traffic Management System**

**Scenario:** A city’s traffic management department wants to develop a system to manage traffic lights in real-time to reduce congestion.

**Tasks:**

1. Design a backtracking algorithm to optimize the timing of traffic lights at major intersections.

2. Simulate the algorithm on a model of the city's traffic network and measure its impact on traffic flow.

3. Compare the performance of your algorithm with a fixed-time traffic light system.

**Deliverables:**

● Pseudocode and implementation of the traffic light optimization algorithm.

● Simulation results and performance analysis.

● Comparison with a fixed-time traffic light system.

**Reasoning:** Justify the use of backtracking for this problem. Discuss the complexities involved in real-time traffic management and how your algorithm addresses them.

**Code:**

import random

import time

# Constants

NUM\_INTERSECTIONS = 4

DEFAULT\_GREEN\_TIME = 30 # seconds

DEFAULT\_RED\_TIME = 30 # seconds

MAX\_ADJUSTMENT = 10 # maximum adjustment time in seconds

MAX\_ITERATIONS = 10 # max number of iterations for simulation

# Intersection class to hold traffic light states and timings

class Intersection:

def \_init\_(self, id):

self.id = id

self.green\_time = DEFAULT\_GREEN\_TIME

self.red\_time = DEFAULT\_RED\_TIME

self.traffic\_density = 0 # Initially no traffic

def update\_traffic\_density(self):

# Simulate traffic density as a random number between 0 and 100

self.traffic\_density = random.randint(0, 100)

def \_str\_(self):

return (f"Intersection {self.id}: Green Time = {self.green\_time}s, "

f"Red Time = {self.red\_time}s, Traffic Density = {self.traffic\_density}")

# Function to adjust timings based on traffic density

def adjust\_timings(intersections, index=0):

if index == len(intersections):

return True

intersection = intersections[index]

intersection.update\_traffic\_density()

original\_green\_time = intersection.green\_time

original\_red\_time = intersection.red\_time

# Try increasing green time

if intersection.traffic\_density > 50: # High traffic density

intersection.green\_time = min(60, original\_green\_time + MAX\_ADJUSTMENT)

intersection.red\_time = max(0, original\_red\_time - MAX\_ADJUSTMENT)

if adjust\_timings(intersections, index + 1):

return True

# Backtrack and reset

intersection.green\_time = original\_green\_time

intersection.red\_time = original\_red\_time

# Try decreasing green time

if intersection.traffic\_density < 20: # Low traffic density

intersection.green\_time = max(10, original\_green\_time - MAX\_ADJUSTMENT)

intersection.red\_time = min(60, original\_red\_time + MAX\_ADJUSTMENT)

if adjust\_timings(intersections, index + 1):

return True

# Backtrack and reset

intersection.green\_time = original\_green\_time

intersection.red\_time = original\_red\_time

# If no suitable adjustment is found, return False

return False

# Initialize intersections

intersections = [Intersection(i) for i in range(NUM\_INTERSECTIONS)]

# Simulation loop

for iteration in range(MAX\_ITERATIONS): # Run the simulation for a number of iterations

print(f"\nIteration {iteration + 1}: Adjusting traffic lights based on traffic conditions...\n")

if not adjust\_timings(intersections):

print("No suitable adjustment found. Keeping current timings.")

for intersection in intersections:

print(intersection)

time.sleep(1) # Pause for a second before the next adjustment (simulates real-time adjustment)

print("\nSimulation finished.")

**Time Complexity**: O(MAX\_ITERATIONS×2NUM\_INTERSECTIONS)

**Space Complexity**: O(NUM\_INTERSECTIONS)