DAA ASSIGNMENT DOC-1

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.

def dijkstra(graph, s):

d = {node: float('inf') for node in graph}

d[s] = 0

uv = list(graph.keys())

while uv:

mind = float('inf')

minn = None

for node in uv:

if d[node] < mind:

mind = d[node]

minn = node

uv.remove(minn)

for n, w in graph[minn].items():

ndist= d[minn] + w

if ndist < d[n]:

d[n] = ndist

return d

graph = {

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

'B': {'D': 2, 'E': 3},

'C': {'D': 3, 'E': 5},

'D': {'E': 1},

'E': {}

}

source = 'A'

dist = dijkstra(graph, source)

print("Distance from source", source)

for n, d in dist.items():

print(n, ":",d)

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 p

def dynamic\_pricing(products, time\_periods, inventory, competitor\_prices, demand\_elasticity):

optimal\_prices = {product: [0] \* len(time\_periods) for product in products}

for t in range(len(time\_periods) - 1, -1, -1):

for product in products:

max\_revenue = 0

optimal\_price = 0

for price in range(1, 101):

revenue = price \* (inventory[product] - (price - competitor\_prices[product][t]) \* demand\_elasticity[product])

if revenue > max\_revenue:

max\_revenue = revenue

optimal\_price = price

optimal\_prices[product][t] = optimal\_price

inventory[product] -= optimal\_price - competitor\_prices[product][t]

return optimal\_prices

products = ["P1", "P2", "P3"]

time\_periods = ["Day 1", "Day 2", "Day 3", "Day 4", "Day 5"]

inventory = {"P1": 100, "P2": 80, "P3": 90}

competitor\_prices = {

"P1": [20, 22, 18, 21, 19],

"P2": [25, 23, 27, 24, 26],

"P3": [18, 20, 19, 22, 21]

}

demand\_elasticity = {

"P1": 0.8,

"P2": 0.6,

"P3": 0.9

}

optimal\_prices = dynamic\_pricing(products, time\_periods, inventory, competitor\_prices, demand\_elasticity)

print("Optimal Pricing Strategy:")

for product, prices in optimal\_prices.items():

print(f"{product}: {', '.join(map(str, prices))}")

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.

import numpy as np

def create\_adjacency\_matrix(edges):

n = max(max(edge) for edge in edges) + 1

adj\_matrix = np.zeros((n, n))

for u, v in edges:

adj\_matrix[u][v] = 1

return adj\_matrix

def pagerank(adj\_matrix, damping\_factor=0.85, max\_iterations=100, tolerance=1e-6):

n = len(adj\_matrix)

pr = np.ones(n) / n

for \_ in range(max\_iterations):

new\_pr = np.ones(n) \* (1 - damping\_factor) / n

for i in range(n):

for j in range(n):

if adj\_matrix[j][i] == 1:

new\_pr[i] += damping\_factor \* pr[j] / sum(adj\_matrix[j])

if np.sum(np.abs(new\_pr - pr)) < tolerance:

break

pr = new\_pr

return pr

def degree\_centrality(adj\_matrix):

return np.sum(adj\_matrix, axis=1)

# Example usage

edges = [(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 4)]

adj\_matrix = create\_adjacency\_matrix(edges)

pagerank\_scores = pagerank(adj\_matrix)

degree\_scores = degree\_centrality(adj\_matrix)

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

print("Degree Centrality Scores:", degree\_scores)

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

def detect\_fraud(transactions, rules):

fraudulent\_transactions = []

for transaction in transactions:

is\_fraudulent = any(rule(transaction) for rule in rules)

if is\_fraudulent:

fraudulent\_transactions.append(transaction["id"])

return fraudulent\_transactions

def rule1(transaction):

return transaction["amount"] > 10000

def rule2(transaction):

return len(set(transaction["location"] for transaction in transactions)) > 3

def evaluate\_performance(actual\_fraudulent, predicted\_fraudulent):

true\_positives = len(set(actual\_fraudulent) & set(predicted\_fraudulent))

false\_positives = len(set(predicted\_fraudulent) - set(actual\_fraudulent))

false\_negatives = len(set(actual\_fraudulent) - set(predicted\_fraudulent))

precision = true\_positives / (true\_positives + false\_positives) if (true\_positives + false\_positives) > 0 else 0

recall = true\_positives / (true\_positives + false\_negatives) if (true\_positives + false\_negatives) > 0 else 0

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

return {"precision": precision, "recall": recall, "f1\_score": f1\_score}

def improve\_algorithm(transactions, rules):

return detect\_fraud(transactions, rules)

transactions = [

{"id": 1, "amount": 5000, "location": "New York"},

{"id": 2, "amount": 15000, "location": "Los Angeles"},

{"id": 3, "amount": 8000, "location": "Chicago"},

{"id": 4, "amount": 12000, "location": "New York"},

{"id": 5, "amount": 7000, "location": "Los Angeles"},

{"id": 6, "amount": 20000, "location": "San Francisco"},

{"id": 7, "amount": 9000, "location": "Chicago"},

{"id": 8, "amount": 6000, "location": "New York"},

{"id": 9, "amount": 18000, "location": "Los Angeles"},

{"id": 10, "amount": 11000, "location": "Chicago"}

]

rules = [rule1, rule2]

fraudulent\_transactions = detect\_fraud(transactions, rules)

print("Potentially Fraudulent Transactions:", fraudulent\_transactions)

actual\_fraudulent = [2, 4, 6, 9]

performance\_metrics = evaluate\_performance(actual\_fraudulent, fraudulent\_transactions)

print("Performance Metrics:")

print(performance\_metrics)

improved\_fraudulent\_transactions = improve\_algorithm(transactions, rules)

print("Improved Potentially Fraudulent Transactions:", improved\_fraudulent\_transactions)