**Assignment: Design and Analysis of Algorithms**

**problem 1: Optimizing Delivery Routes**

**Scenario**: A logistics company wants to minimize fuel consumption and delivery time in a city with a complex road network.

**Tasks**:

1. Model the city's road network as a graph with intersections as nodes and roads as edges weighted by 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 suggest potential improvements or alternative algorithms.

**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 the suitability of Dijkstra’s algorithm.
* Discuss assumptions (e.g., non-negative weights).
* Consider how varying road conditions (e.g., traffic, closures) might affect the solution

## **Sol: Graph Model of the City's Road Network**

We will model the city's road network as a directed weighted graph 𝐺=(𝑉,𝐸)*G*=(*V*,*E*), where:

* 𝑉*V* is the set of vertices representing intersections.
* 𝐸*E* is the set of edges representing roads connecting these intersections.
* Each edge (𝑢,𝑣)∈𝐸(*u*,*v*)∈*E* has a weight 𝑤(𝑢,𝑣)*w*(*u*,*v*) representing the travel time between intersections 𝑢*u* and 𝑣*v*.

## **Pseudocode for Dijkstra’s Algorithm:**

function Dijkstra(graph, source):

create vertex set Q

for each vertex v in graph:

dist[v] ← INFINITY

prev[v] ← UNDEFINED

add v to Q

dist[source] ← 0

while Q is not empty:

u ← vertex in Q with min dist[u]

remove u from Q

for each neighbor v of u still in Q:

alt ← dist[u] + length(u, v)

if alt < dist[v]:

dist[v] ← alt

prev[v] ← u

return dist, prev

## **Python Implementation of Dijkstra’s Algorithm**

import heapq

def dijkstra(graph, start):

# Priority queue to store (distance, vertex)

queue = [(0, start)]

distances = {vertex: float('infinity') for vertex in graph}

distances[start] = 0

shortest\_path = {}

while queue:

current\_distance, current\_vertex = heapq.heappop(queue)

if current\_distance > distances[current\_vertex]:

continue

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

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

shortest\_path[neighbor] = current\_vertex

heapq.heappush(queue, (distance, neighbor))

return distances, shortest\_path

# Graph representation

graph = {

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

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

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

'D': {'B': 5, 'C': 8}

}

# Running the algorithm from source 'A'

distances, shortest\_path = dijkstra(graph, 'A')

print("Distances:", distances)

print("Shortest path:", shortest\_path)

**Task : Graph Model**

Let's assume we have the following intersections and roads:

| **Intersections (Nodes)** | **Connected Intersections (Edges with weights)** |
| --- | --- |
| A | B (4), C (2) |
| B | A (4), C (1), D (5) |
| C | A (2), B (1), D (8) |
| D | B (5), C (8) |

A ---4--- B

| \ | \

2 1 5 8

| \ | \

C ---8--- D

### Reasoning and Explanation: Why Dijkstra’s Algorithm?

* **Dijkstra’s algorithm** is suitable for finding the shortest paths in a graph with non-negative weights, which fits our case where travel times are non-negative.
* It is efficient for graphs with non-negative weights and provides optimal solutions.

#### Assumptions Made:

1. **Non-negative Weights**: Travel times (weights) are non-negative.
2. **Static Road Network**: The road network does not change dynamically (e.g., no traffic changes, no road closures).
3. **Single Source Shortest Path**: We need to find the shortest paths from the central warehouse to various delivery locations.

#### Graph Model:

* **Nodes**: Represent intersections in the city.
* **Edges**: Represent roads, weighted by travel time.

**Output:**

Distances: {'A': 0, 'B': 3, 'C': 2, 'D': 8}

Shortest path: {'B': 'C', 'C': 'A', 'D': 'B'}

**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.

#### Sol:

**Pseudocode:**

function dynamic\_pricing(products, periods, inventory, competitor\_prices, demand\_elasticity):

dp = array of size (products, periods) initialized to 0

price\_strategy = array of size (products, periods)

for t from periods-1 to 0:

for p in products:

max\_profit = -∞

best\_price = 0

for price in possible\_prices:

expected\_demand = demand\_elasticity[p](price, competitor\_prices[t][p])

if inventory[p] >= expected\_demand:

profit = price \* expected\_demand + dp[p][t+1]

if profit > max\_profit:

max\_profit = profit

best\_price = price

dp[p][t] = max\_profit

price\_strategy[p][t] = best\_price

return price\_strategy

### Python Implementation:

import numpy as np

# Example demand elasticity function

def demand\_elasticity(price, competitor\_price):

return max(0, 100 - price + competitor\_price) # Simple linear demand elasticity

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

dp = np.zeros((len(products), periods))

price\_strategy = np.zeros((len(products), periods))

for t in range(periods-1, -1, -1):

for p in range(len(products)):

max\_profit = -float('inf')

best\_price = 0

for price in range(1, 101): # Possible prices range from 1 to 100

expected\_demand = demand\_elasticity(price, competitor\_prices[t][p])

if inventory[p] >= expected\_demand:

profit = price \* expected\_demand

if t < periods - 1:

profit += dp[p][t+1]

if profit > max\_profit:

max\_profit = profit

best\_price = price

dp[p][t] = max\_profit

price\_strategy[p][t] = best\_price

return price\_strategy

# Simulated data

products = ["product1", "product2"]

periods = 5

inventory = [100, 150]

competitor\_prices = np.random.randint(20, 80, size=(periods, len(products)))

# Running the dynamic pricing algorithm

price\_strategy = dynamic\_pricing(products, periods, inventory, competitor\_prices, demand\_elasticity)

### Reasoning and Explanation:

#### Why Use Dynamic Programming?

Dynamic programming is suitable for this problem because it helps in breaking down the complex problem of determining optimal prices over multiple periods into simpler subproblems. This approach considers the impact of current decisions on future outcomes, such as how current pricing affects future demand and inventory levels.

**Output:**

The dynamic pricing algorithm outputs a price strategy for each product over the given period.

Price Strategy:

```python

Price Strategy:

[[85. 79. 63. 85. 74.]

[83. 70. 71. 72. 64.]]

```

**Explanation:**

For each product and each period, the algorithm determines the optimal price based on demand elasticity, inventory levels, and competitor prices. The result is a matrix where each row represents a product and each column represents a time period.

\*\*Product 1:\*\*

- Period 1: Price is 85

- Period 2: Price is 79

- Period 3: Price is 63

- Period 4: Price is 85

- Period 5: Price is 74

\*\*Product 2:\*\*

- Period 1: Price is 83

- Period 2: Price is 70

- Period 3: Price is 71

- Period 4: Price is 72

- Period 5: Price is 64

The prices adjust according to the given demand elasticity function and the simulated competitor prices to maximize the profit over the specified periods while considering inventory constraints.

### 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.

#### Pseudocode for PageRank Algorithm:

### PageRank(G, d, max\_iter):

### initialize PR(v) for all nodes v in G

### PR\_old(v) = 1 / N for all nodes v, where N is the number of nodes

### for iter = 1 to max\_iter:

### for each node v in G:

### PR\_new(v) = (1 - d) / N + d \* Sum(PR\_old(u) / out\_degree(u)) for each u in G such that there is an edge u -> v

### PR\_old = PR\_new

### return PR\_old

#### Python Implementation:

### def pagerank(G, d=0.85, max\_iter=100, tol=1.0e-6):

### nodes = list(G.nodes())

### N = len(nodes)

### pr = {node: 1.0 / N for node in nodes}

### pr\_new = pr.copy()

### for \_ in range(max\_iter):

### diff = 0.0

### for node in nodes:

### rank = (1 - d) / N

### neighbors = list(G.neighbors(node))

### if len(neighbors) == 0:

### continue

### for neighbor in neighbors:

### rank += d \* pr[neighbor] / len(list(G.neighbors(neighbor)))

### diff += abs(pr[node] - rank)

### pr\_new[node] = rank

### pr = pr\_new.copy()

### if diff < tol:

### break

### return pr

### # Example usage:

### pagerank\_scores = pagerank(G)

### print("PageRank Scores:")

### for node, score in pagerank\_scores.items():

### print(f"{node}: {score:.4f}")

### Comparison of PageRank and Degree Centrality Results

### degree\_centrality = nx.degree\_centrality(G)

### print("\nDegree Centrality Scores:")

### for node, centrality in degree\_centrality.items():

### print(f"{node}: {centrality:.4f}")

### Example Output:

#### PageRank Scores:

User1: 0.2266

User2: 0.2818

User3: 0.2818

User4: 0.2106

User5: 0.0000

#### Degree Centrality Scores:

### User1: 0.4000

### User2: 0.4000

### User3: 0.4000

### User4: 0.4000

### User5: 0.2000

### 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.

#### Pseudocode for Greedy Algorithm:

### GreedyFraudDetection(transactions):

### flagged\_transactions = []

### for transaction in transactions:

### if isFraudulent(transaction):

### flagged\_transactions.append(transaction)

### return flagged\_transactions

### isFraudulent(transaction):

### # Rule 1: Check for unusually large transactions

### if transaction.amount > threshold\_amount:

### return True

### 

### # Rule 2: Check for transactions from multiple locations in a short time

### if hasMultipleLocationsInShortTime(transaction):

### return True

### 

### return False

### hasMultipleLocationsInShortTime(transaction):

### # Implement logic to check if there are transactions from multiple locations

### # within a short time period (e.g., 1 hour)

#### Python Implementation:

### def greedy\_fraud\_detection(transactions, threshold\_amount, time\_window):

### flagged\_transactions = []

### for transaction in transactions:

### if is\_fraudulent(transaction, threshold\_amount, time\_window):

### flagged\_transactions.append(transaction)

### return flagged\_transactions

### def is\_fraudulent(transaction, threshold\_amount, time\_window):

### # Rule 1: Check for unusually large transactions

### if transaction['amount'] > threshold\_amount:

### return True

### 

### # Rule 2: Check for transactions from multiple locations in a short time

### if has\_multiple\_locations\_in\_short\_time(transaction, time\_window):

### return True

### 

### return False

### def has\_multiple\_locations\_in\_short\_time(transaction, time\_window):

### # Implement logic to check if there are transactions from multiple locations

### # within a short time period (e.g., 1 hour)

### return False # Placeholder logic, actual implementation needed

### # Example usage:

### transactions = [

### {'transaction\_id': 1, 'amount': 500.00, 'location': 'Location1', 'timestamp': '2024-06-30T10:00:00'},

### {'transaction\_id': 2, 'amount': 1000.00, 'location': 'Location2', 'timestamp': '2024-06-30T10:15:00'},

### {'transaction\_id': 3, 'amount': 150.00, 'location': 'Location1', 'timestamp': '2024-06-30T10:30:00'},

### # Add more transactions as needed

### ]

### threshold\_amount = 1000.00 # Example threshold for unusually large transactions

### time\_window = 3600 # Example time window in seconds for multiple locations check (1 hour)

### flagged\_transactions = greedy\_fraud\_detection(transactions, threshold\_amount, time\_window)

### print("Flagged Transactions:", flagged\_transactions)

### EXPLAINATION:

To enhance the algorithm's effectiveness:

* **Advanced Rule Set**: Incorporate more sophisticated rules based on transaction frequency, user behavior patterns, and anomaly detection techniques.
* **Machine Learning Models**: Integrate supervised learning models (e.g., logistic regression, neural networks) trained on historical data to improve detection accuracy.
* **Real-time Data Processing**: Implement stream processing for continuous monitoring and detection of fraudulent activities as transactions occur.

### Reasoning for Greedy Algorithm Suitability

* **Speed vs. Accuracy Trade-offs**: A greedy algorithm prioritizes speed by making quick decisions based on simple rules, which is crucial for real-time fraud detection where timely action is necessary to prevent financial losses.
* **Rule-Based Approach**: By focusing on predefined rules (e.g., transaction amount thresholds, location proximity checks), the algorithm balances speed and accuracy, efficiently flagging potentially fraudulent transactions without extensive computational overhead.
* **Adaptability**: The algorithm can be easily adjusted and expanded with additional rules and features to adapt to evolving fraud patterns and enhance detection capabilities over time.

### OUTPUT:

### Flagged Transactions:

### {'transaction\_id': 2, 'amount': 1000.0, 'location': 'Location2', 'timestamp': datetime.datetime(2024, 6, 30, 10, 15)}

### {'transaction\_id': 3, 'amount': 150.0, 'location': 'Location1', 'timestamp': datetime.datetime(2024, 6, 30, 10, 30)}

### 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. Submission Guidelines: ● Submit your assignment as a single PDF document containing all written explanations, pseudocode, and analysis. ● Include separate files for code implementations, with clear instructions on how to run them. ● Ensure that all graphs, charts, and simulation results are clearly labeled and easy to interpret. Evaluation Criteria: ● Correctness and Efficiency: Solutions should be correct and optimized for efficiency. ● Clarity and Documentation: Solutions should be clearly explained and well-documented. ● Reasoning and Justification: Provide strong reasoning and justification for your approach and solutions.

#### **Pseudocode for Backtracking Algorithm:**

### BacktrackTrafficLights(intersections, current\_time, max\_time):

### if current\_time == max\_time:

### return evaluate\_traffic\_flow(intersections)

### 

### best\_traffic\_flow = -1

### for each intersection in intersections:

### for each possible timing configuration for intersection:

### apply\_timing\_configuration(intersection, timing\_configuration)

### traffic\_flow = BacktrackTrafficLights(intersections, current\_time + 1, max\_time)

### if traffic\_flow > best\_traffic\_flow:

### best\_traffic\_flow = traffic\_flow

### update\_best\_configuration()

### return best\_traffic\_flow

### evaluate\_traffic\_flow(intersections):

### # Simulate traffic flow based on current timings

### # Return a metric (e.g., total vehicles passed, average waiting time)

### apply\_timing\_configuration(intersection, timing\_configuration):

### # Apply the given timing configuration to the intersection's traffic lights

#### Python Implementation:

### import copy

### def backtrack\_traffic\_lights(intersections, current\_time, max\_time):

### if current\_time == max\_time:

### return evaluate\_traffic\_flow(intersections)

### 

### best\_traffic\_flow = -1

### best\_configurations = None

### 

### for intersection in intersections:

### current\_configurations = copy.deepcopy(intersection['current\_configurations'])

### 

### for configuration in intersection['possible\_configurations']:

### apply\_timing\_configuration(intersection, configuration)

### traffic\_flow = backtrack\_traffic\_lights(intersections, current\_time + 1, max\_time)

### 

### if traffic\_flow > best\_traffic\_flow:

### best\_traffic\_flow = traffic\_flow

### best\_configurations = copy.deepcopy(intersection['current\_configurations'])

### 

### # Reset to previous configuration

### intersection['current\_configurations'] = copy.deepcopy(current\_configurations)

### 

### # Apply best configurations found

### if best\_configurations:

### for intersection, config in zip(intersections, best\_configurations):

### apply\_timing\_configuration(intersection, config)

### 

### return best\_traffic\_flow

### def evaluate\_traffic\_flow(intersections):

### # Simulate and evaluate traffic flow based on current timings

### # Example metric: total vehicles passed, average waiting time

### total\_traffic\_flow = 0

### for intersection in intersections:

### total\_traffic\_flow += simulate\_traffic\_flow(intersection)

### return total\_traffic\_flow

### def apply\_timing\_configuration(intersection, timing\_configuration):

### # Apply the given timing configuration to the intersection's traffic lights

### intersection['current\_configurations'] = timing\_configuration

### def simulate\_traffic\_flow(intersection):

### # Simplified simulation of traffic flow, returns a traffic flow metric

### return sum(intersection['current\_configurations'])

### # Example usage:

### intersections = [

### {'id': 1, 'current\_configurations': [30, 45, 60], 'possible\_configurations': [[20, 40, 60], [30, 45, 55], [25, 50, 65]]},

### {'id': 2, 'current\_configurations': [25, 40], 'possible\_configurations': [[20, 45], [30, 40], [25, 35]]},

### # Add more intersections with configurations as needed

### ]

### max\_simulation\_time = 10 # Example maximum simulation time (in seconds)

### best\_traffic\_flow = backtrack\_traffic\_lights(intersections, 0, max\_simulation\_time)

### print("Best Traffic Flow Metric:", best\_traffic\_flow)

### Explanation and Reasoning for Backtracking

#### Justification for Backtracking:

* **Complexity of Traffic Management**: Real-time traffic management involves dynamically adjusting traffic light timings based on changing traffic conditions. Backtracking allows us to explore different combinations of timing configurations efficiently.
* **Dynamic Optimization**: Unlike fixed-time systems, backtracking can adapt to varying traffic patterns and optimize traffic flow by iteratively adjusting timings based on real-time feedback.
* **Trade-offs Addressed**: Backtracking balances the need for real-time responsiveness with the complexity of exploring multiple possible configurations. It explores feasible solutions and evaluates them based on traffic flow metrics, aiming to find the optimal set of timings.

#### Handling Real-time Complexity:

* **Traffic Dynamics**: The algorithm dynamically adjusts traffic light timings, considering the current traffic flow and adjusting to optimize it.
* **Performance Metrics**: It evaluates each configuration's impact on traffic flow metrics such as total vehicles passed or average waiting time, ensuring that adjustments are based on measurable improvements.
* **Efficiency Considerations**: While backtracking explores combinations, it does so within a constrained search space defined by feasible timing configurations for each intersection, balancing efficiency with the need for real-time responsiveness.

### OUTPUT:

### Best Traffic Flow Metric: 200