Assignment: Design and Analysis of Algorithms

NAME: GUNALAN V

REG NO: 192325038

## 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. **Modeling the City's Road Network**
   * **Graph Representation**: The city’s road network is represented as a graph with intersections as nodes and roads as edges. The weights on the edges represent travel time.
   * **Example Graph Model**:

plaintext

Copy code

Intersections: A, B, C, D

Roads: A-B, A-C, B-D, C-D

Weights (travel time): A-B: 10, A-C: 15, B-D: 12, C-D: 10

1. **Implementing Dijkstra’s Algorithm**
   * **Pseudocode**:

plaintext

Copy code

function Dijkstra(Graph, source):

dist[source] ← 0

for each vertex v in Graph:

if v ≠ source:

dist[v] ← ∞

add v to priority queue Q

while Q is not empty:

u ← vertex in Q with min dist[u]

remove u from Q

for each neighbor v of u:

alt ← dist[u] + length(u, v)

if alt < dist[v]:

dist[v] ← alt

update priority queue Q

return dist

* + **Python Implementation**:

python

Copy code

import heapq

def dijkstra(graph, source):

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

dist[source] = 0

priority\_queue = [(0, source)]

while priority\_queue:

current\_dist, current\_node = heapq.heappop(priority\_queue)

if current\_dist > dist[current\_node]:

continue

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

distance = current\_dist + weight

if distance < dist[neighbor]:

dist[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return dist

# Example graph

graph = {

'A': {'B': 10, 'C': 15},

'B': {'A': 10, 'D': 12},

'C': {'A': 15, 'D': 10},

'D': {'B': 12, 'C': 10}

}

source = 'A'

print(dijkstra(graph, source))

1. **Analyzing the Algorithm’s Efficiency**
   * **Time Complexity**: O((V + E) log V), where V is the number of vertices and E is the number of edges.
   * **Space Complexity**: O(V + E).
   * **Potential Improvements**: Consider using the A\* algorithm for improved performance in certain cases, particularly where heuristic information is available.

### Reasoning

Dijkstra's algorithm is suitable for this problem because it efficiently finds the shortest paths in a graph with non-negative edge weights. Assumptions include non-negative weights and static road conditions, which might not reflect real-time traffic changes.

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

### Scenario

An e-commerce company wants to implement a dynamic pricing algorithm to adjust prices in real-time based on demand and competitor prices.

### Tasks

1. **Designing a Dynamic Programming Algorithm**
   * **Approach**: Develop a dynamic programming approach to adjust prices in real-time considering factors such as inventory levels, competitor pricing, and demand elasticity.
   * **Pseudocode**:

plaintext

Copy code

function DynamicPricing(products, period):

dp\_table ← initialize table with zeros

for each product in products:

for each day in period:

calculate optimal price based on previous prices, demand, and competitor pricing

update dp\_table

return dp\_table

1. **Simulation Results**
   * **Comparison**: Compare the performance of the dynamic pricing algorithm with a static pricing strategy using simulated data.
   * **Analysis**: Discuss the benefits of dynamic pricing, such as increased revenue and better inventory management, and the drawbacks, including the complexity of implementation and the need for real-time data.

### Reasoning

Dynamic programming is chosen for this problem due to its ability to optimize decisions over time. The complexity of incorporating factors like demand elasticity and competitor pricing justifies this approach.

## 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. **Modeling the Social Network**
   * **Graph Representation**: Users are represented as nodes and their connections as edges.
2. **Implementing the PageRank Algorithm**
   * **Pseudocode**:

plaintext

Copy code

function PageRank(Graph, damping\_factor, iterations):

initialize ranks for all nodes

for i from 1 to iterations:

for each node in Graph:

rank ← calculate rank based on in-links and damping factor

return ranks

1. **Comparison with Degree Centrality**
   * **Analysis**: Compare PageRank with degree centrality and discuss the scenarios where each measure is preferred.

### Reasoning

PageRank is effective for identifying influential users because it considers both the quantity and quality of connections, unlike degree centrality, which only considers the number of direct connections.

## 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. **Designing a Greedy Algorithm**
   * **Approach**: Design a greedy algorithm that flags transactions based on predefined rules like transaction amount and location.
   * **Pseudocode**:

plaintext

Copy code

function FraudDetection(transactions, rules):

flagged\_transactions ← []

for each transaction in transactions:

if transaction meets any rule in rules:

add to flagged\_transactions

return flagged\_transactions

1. **Performance Evaluation**
   * **Metrics**: Use precision, recall, and F1 score to evaluate the algorithm’s performance on historical data.
2. **Suggestions for Improvement**
   * **Enhancements**: Consider machine learning-based approaches for better detection accuracy.

### Reasoning

A greedy algorithm is suitable for real-time detection due to its speed, though it may trade off some accuracy for faster decision-making.

## 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. **Designing a Backtracking Algorithm**
   * **Approach**: Use backtracking to optimize traffic light timings at major intersections.
   * **Pseudocode**:

plaintext

Copy code

function OptimizeTrafficLights(traffic\_network):

if no conflicts in current configuration:

return current configuration

for each possible configuration:

if configuration is valid:

recursively try to find the optimal configuration

return best configuration

1. **Simulation Results**
   * **Impact on Traffic Flow**: Simulate the algorithm and measure its impact on traffic flow.
2. **Comparison with Fixed-Time System**
   * **Performance**: Compare the backtracking approach with a fixed-time traffic light system and discuss the benefits in terms of reduced congestion.

### Reasoning

Backtracking is justified for this problem due to the complexity and real-time requirements of traffic management. It allows for exploring multiple configurations to find the optimal solution.