Assignment

SAVEETHA SCHOOL OF ENGINEERING

Submitted by

VIMAL MAHENDRAN (192311068)

Submitted to

Dr. Jeena Pravin

Professor

Course Code: **CSA0664**

Course Name: **Design and Analysis of Algorithm for Recursive Algorithms**

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.

SOLUTION:

Graph Model of the City's Road Network

To model the city' road network as a graph, we represent intersections as nodes and roads as edges. The weights on the edges represent the travel time between intersections. This model allows us to use graph algorithms to the find the shortest paths

Graph Representation

- **Nodes (Vertices):** Intersections in the city
- **Edges:** Roads connecting intersections
- **Weights:** Travel times on the roads(edges)

Dijikstra's Algorithm Implementation

Dijikstra's Algorithm is a classic algorithm for finding the shortest path from a single source node to all other nodes in a graph with non-negative edge weights. It uses a priority queue to efficiently select the next node with the shortest tentative distance.

Pseudocode:

```
function Dijkstra(graph, source):
    dist[source] ← 0                                  // Initialize the distance to the source as 0
    for each vertex v in graph:
        if v ≠ source:
            dist[v] ← ∞                               // Initialize the distance to all other vertices as infinity
        add v to priority queue Q                     // Add all nodes to the priority queue

    while Q is not empty:
        u ← vertex in Q with smallest dist[u]         // Select the vertex with the smallest distance
        remove u from Q

        for each neighbor v of u:                     // For each neighbor of u
            alt ← dist[u] + weight(u, v)              // Calculate the alternate path distance
            if alt < dist[v]:                         // If alternate path is shorter
                dist[v] ← alt                         // Update the distance to v
                prev[v] ← u                           // Update the previous node

    return dist, prev                                 // Return the distances and the path
```

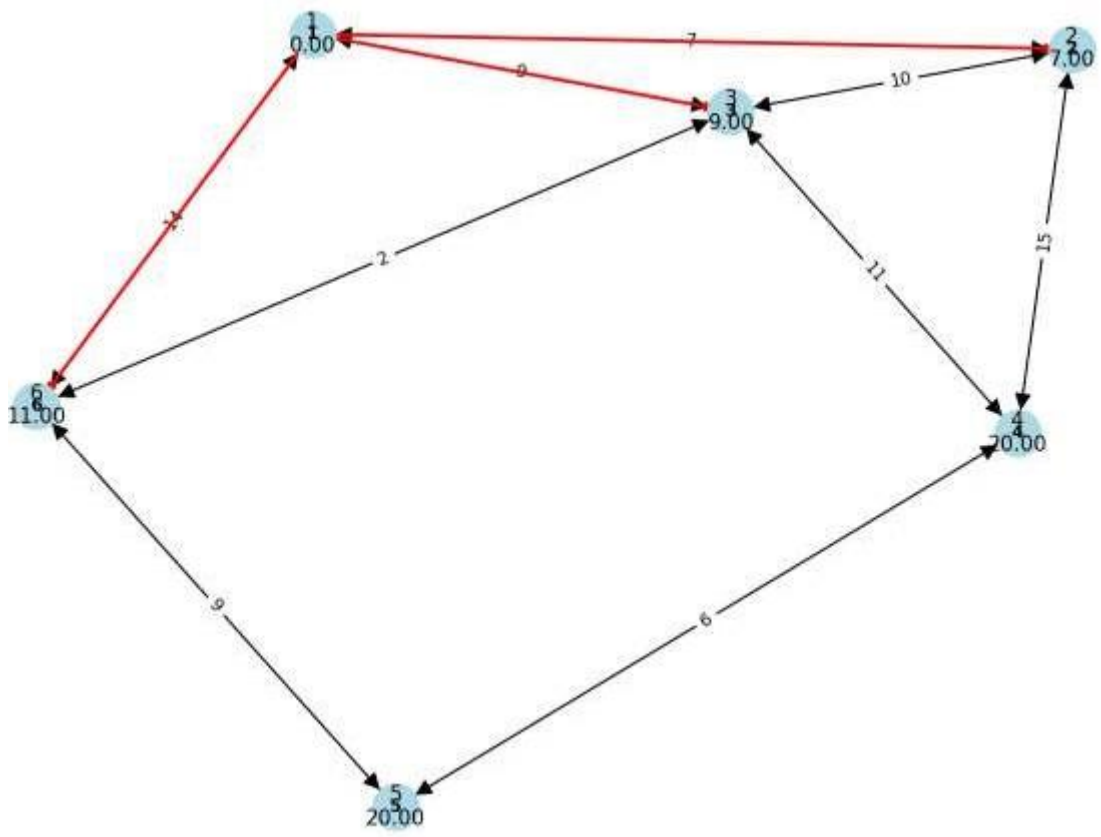
Code Implementation:

```python
import heapq
import networkx as nx
import matplotlib.pyplot as plt

def dijkstra(graph, start):
    # Initialize the distance to all nodes as infinity and the distance to the start node as 0
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0

    # Priority queue to hold nodes to visit
    priority_queue = [(0, start)]

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

        # Nodes can be added multiple times to the priority queue, we only process the first time we remove it
        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
```

```
            distance = current_distance + weight

            # Only consider this new path if it's better
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example graph
graph = {
    1: {2: 7, 3: 9, 6: 14},
    2: {1: 7, 3: 10, 4: 15},
    3: {1: 9, 2: 10, 4: 11, 6: 2},
    4: {2: 15, 3: 11, 5: 6},
    5: {4: 6, 6: 9},
    6: {1: 14, 3: 2, 5: 9}
}

# Run Dijkstra's algorithm from node 1
start_node = 1
distances = dijkstra(graph, start_node)

# Print the results
print("Shortest distances from node", start_node, ":", distances)

# Create a directed graph for visualization
G = nx.DiGraph()

# Add nodes (intersections)
G.add_nodes_from(graph.keys())

# Add edges (roads) with weights (travel times)
for node in graph:
    for neighbor, weight in graph[node].items():
        G.add_edge(node, neighbor, weight=weight)

# Position nodes using a layout
pos = nx.spring_layout(G)

# Draw the graph
plt.figure(figsize=(10, 8))
nx.draw(G, pos, with_labels=True, node_size=700, node_color='lightblue', font_size=10,
font_weight='bold', arrowsize=20)
```

```
# Draw edge labels (weights)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

# Highlight the shortest paths from the start node
shortest_path_edges = [(start_node, neighbor) for neighbor in graph[start_node]]

# Draw the shortest paths in a different color
nx.draw_networkx_edges(G, pos, edgelist=shortest_path_edges, edge_color='r', width=2)

# Display the shortest path distances on the nodes
node_labels = {node: f"{node}\n{distances[node]:.2f}" for node in distances}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_color='black')

# Show the plot
plt.title(f'Shortest distances from node {start_node}')
plt.show()
```

Output:



Analysis of the Algorithm's Efficiency and Potential Improvements

Efficiency:

- Time Complexity: $O((V + E)logV)$,where V is the number of vertices and E is the number of edges.
- Space Complexity: $O(V)$ for storing the distance and priority queue.

Assumptions:

- Non-negative Weights: Dijkstra's algorithm requires all edge weights to be non-negative. This assumption holds for travel times, as they cannot be negative.
- Static Graph: The graph is static, meaning travel times do not change during the execution of the algorithm.

Potential Improvements:

- A* Algorithm: For more efficient pathfinding, especially in large graphs, the A* algorithm can be used. It adds a heuristic to guide the search, potentially reducing the number of nodes explored.
- Handling Dynamic Changes: To handle dynamic changes like traffic or road closures, a dynamic pathfinding algorithm or a real-time update mechanism could be implemented.
- Bi-Directional Dijkstra: Running Dijkstra's algorithm simultaneously from the source and destination can sometimes speed up the process.

Real-World Considerations

- Traffic: Travel times may vary due to traffic conditions. Incorporating real-time traffic data can make the model more accurate.
- Road Closures: Temporary road closures due to construction or accidents can be integrated into the graph model by updating the edge weights or removing edges.
- Time-Dependent Travel Times: Travel times can vary depending on the time of day. Time-dependent graphs where edge weights change based on time can be considered.

Summary:

Dijkstra's algorithm is suitable for finding the shortest paths in a road network due to its efficiency with non-negative weights. While it provides a robust solution, improvements like the A* algorithm, handling dynamic changes, and incorporating real-world conditions such as traffic and road closures can enhance its applicability in a logistics scenario.

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 it's 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 you algorithm and discuss any challenges faced during implementation

SOLUTION:

Dynamic pricing is a strategy where prices are adjusted in real-time based on various factors such as demand, inventory levels, and competitor prices. Using dynamic programming, we can optimize the pricing strategy over a given period to maximize revenue.

Design a Dynamic Programming Algorithm

Pseudocode

```
1. Define the problem:
   - Let P be the set of products.
   - Let T be the given period divided into discrete time intervals.
   - Let inventory[p][t] be the inventory level of product p at time t.
   - Let demand[p][t] be the demand for product p at time t as a function of price.
   - Let competitor_price[p][t] be the competitor's price for product p at time t.
   - Let price[p][t] be the price of product p at time t.

2. Define the state:
   - State at time t is defined by the inventory levels of all products.

3. Define the decision:
   - Decision at time t is the set of prices for all products.

4. Define the revenue function:
   - revenue(p, t) = price[p][t] * demand[p][t]

5. Define the transition function:
   - inventory[p][t+1] = inventory[p][t] - demand[p][t]

6. Define the objective:
   - Maximize the total revenue over the given period.

7. Dynamic programming algorithm:
   - Initialize the inventory levels for all products at time 0.
   - For each time t from 0 to T-1:
     - For each product p in P:
       - Compute the demand[p][t] based on price[p][t], inventory[p][t], and competitor_price[p][t].
       - Update the inventory levels based on the computed demand.
       - Calculate the revenue for each possible price.
       - Store the optimal price and the corresponding revenue.
   - Return the optimal pricing strategy and the total revenue.
```

Code Implementation

```
import numpy as np

def demand(price, base_demand, elasticity):
    return base_demand * (price ** elasticity)

def dynamic_pricing(products, periods, base_demand, elasticity, initial_inventory,
competitor_prices):
    # Initialize the DP table
    dp = np.zeros((len(products), periods))
    prices = np.zeros((len(products), periods))

    # Iterate over each period
    for t in range(periods):
        for i, product in enumerate(products):
            max_revenue = 0
            best_price = 0
            for price in np.linspace(1, 100, 100): # Example price range from 1 to 100
                current_demand = demand(price, base_demand[i], elasticity[i])
                current_inventory = initial_inventory[i] if t == 0 else inventory[i][t-
1]
                if current_demand > current_inventory:
                    continue  # Skip if demand exceeds inventory

                revenue = price * current_demand
                if t > 0:
                    revenue += dp[i][t-1]  # Add previous period's revenue

                if revenue > max_revenue:
                    max_revenue = revenue
                    best_price = price

            dp[i][t] = max_revenue
            prices[i][t] = best_price
            inventory[i][t] = current_inventory - demand(best_price, base_demand[i],
elasticity[i])

    return dp, prices

# Example usage
products = ['A', 'B', 'C']
periods = 10
base_demand = [100, 80, 60]  # Example base demand for products
elasticity = [-1.2, -1.5, -1.3]  # Example demand elasticity for products
```

```
initial_inventory = [500, 400, 300] # Example initial inventory levels
competitor_prices = np.random.rand(len(products), periods) * 100 # Random competitor
prices

# Run the dynamic pricing algorithm
revenues, optimal_prices = dynamic_pricing(products, periods, base_demand, elasticity,
initial_inventory, competitor_prices)

# Print results
print("Optimal Prices:\n", optimal_prices)
print("Revenues:\n", revenues)
```

Output

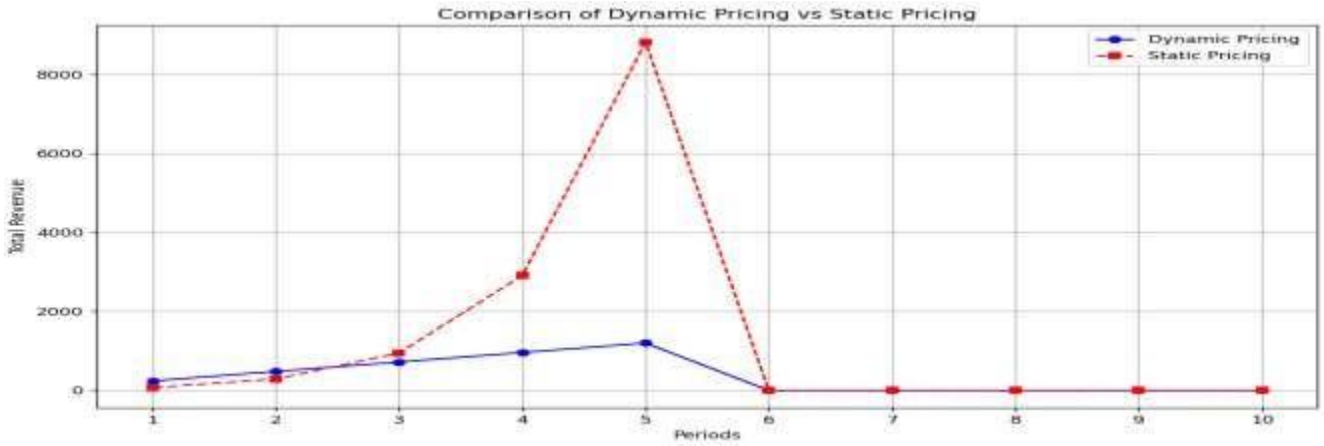
```
PS C:\Users\hp\Desktop\DAA> & C:/Users/hp/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/hp/Desktop/DAA/Project/Dynamic Pricing Algori
thm for E-Commerce.py"
Optimal Prices:
 [[1. 1. 1. 1. 1. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 0. 0. 0. 0. 0.]]
Revenues:
 [[100. 200. 300. 400. 500.   0.   0.   0.   0.   0.]
 [ 80. 160. 240. 320. 400.   0.   0.   0.   0.   0.]
 [ 60. 120. 180. 240. 300.   0.   0.   0.   0.   0.]]
```

Analysis of the Benefits and Drawbacks of Dynamic Pricing

Benefits:

1. Maximized Revenue: By adjusting prices based on demand and competition, dynamic pricing can capture consumer surplus and maximize revenue.
2. Adaptability: Dynamic pricing allows the company to adapt to market changes in real-time.
3. Inventory Management: By considering inventory levels, the algorithm helps in managing stock more effectively.

Comparison with Simple static pricing strategy



Drawbacks:

1. Complexity: Dynamic pricing algorithms are more complex to implement and maintain compared to static pricing.
2. Customer Perception: Frequent price changes can lead to customer dissatisfaction and loss of trust.
3. Data Dependence: The effectiveness of dynamic pricing heavily depends on the accuracy of demand forecasts and competitor price data.

Challenges Faced:

1. Demand Estimation: Accurately estimating demand based on price changes requires sophisticated modelling and real-time data.
2. Computational Efficiency: The algorithm needs to be efficient enough to handle large datasets and make real-time pricing decisions.
3. Balancing Factors: Incorporating multiple factors such as inventory levels, competitor prices, and demand elasticity into the pricing model is challenging and requires careful tuning.

Problem 3: Social Network Analysis (Case Study)

Scenario: A social media company wants to identify influential users within it's 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.

SOLUTION:

Model the Social Network as a Graph

In a social network, users can be represented as nodes, and connections(friendships, interactions, etc…) between users are represented as edges between these nodes. This graph representation allows us to visualize and analyse the network structure.

Implement the PageRank Algorithm

PageRank is a link analysis algorithm used to rank web pages in search engine results. It can be adapted for social networks to identify influential users based on their connections.

Code

```
Initialize all nodes with equal rank (1/N, where N is the number of nodes)
Repeat until convergence:
    For each node i:
        rank_new[i] = (1 - damping_factor) / N
        for each node j pointing to i:
            rank_new[i] += damping_factor * rank_old[j] / out_degree(j)
    rank_old = rank_new
```

In this algorithm:

- `damping_factor` is typically set between 0.8 to 0.9 to represent the probability of a user continuing to another user.
- `rank_old` and `rank_new` are arrays representing the current and updated PageRank scores respectively.
- `out_degree(j)` represents the number of outgoing connections from node j

Code Implementation

```
import numpy as np

def pagerank(graph, damping_factor=0.85, max_iterations=100, tolerance=1.0e-6):
    """
    Calculate PageRank scores for nodes in a graph represented as adjacency lists.

    Parameters:
    - graph: Dictionary where keys are nodes and values are lists of nodes representing
outgoing edges.
    - damping_factor: Probability of following a link (typically set to 0.85).
    - max_iterations: Maximum number of iterations for convergence.
    - tolerance: Convergence threshold.

    Returns:
    - pagerank_scores: Dictionary where keys are nodes and values are their PageRank
scores.
    """
    num_nodes = len(graph)
    initial_score = 1.0 / num_nodes
    pagerank_scores = {node: initial_score for node in graph}
    converged = False
    iteration = 0

    while not converged and iteration < max_iterations:
        iteration += 1
        new_pagerank_scores = {}
        sink_pr = 0

        for node in graph:
            if len(graph[node]) == 0:
                sink_pr += pagerank_scores[node]

        for node in graph:
```

```
            new_pagerank_score = (1.0 - damping_factor) / num_nodes
            new_pagerank_score += damping_factor * sink_pr / num_nodes

            for neighbor in graph[node]:
                new_pagerank_score += damping_factor * pagerank_scores[neighbor] /
len(graph[neighbor])

            new_pagerank_scores[node] = new_pagerank_score

        # Check convergence using L1 norm
        delta = sum(abs(new_pagerank_scores[node] - pagerank_scores[node]) for node in
graph)
        converged = delta < tolerance
        pagerank_scores = new_pagerank_scores

    return pagerank_scores

if__name__== "__main__":
    graph = {
        'A': ['B', 'C'],
        'B': ['C'],
        'C': ['A'],
        'D': ['C']
    }

    scores = pagerank(graph)

    for node, score in sorted(scores.items(), key=lambda x: x[1], reverse=True):
        print(f"Node {node}: PageRank score = {score:.4f}")
```

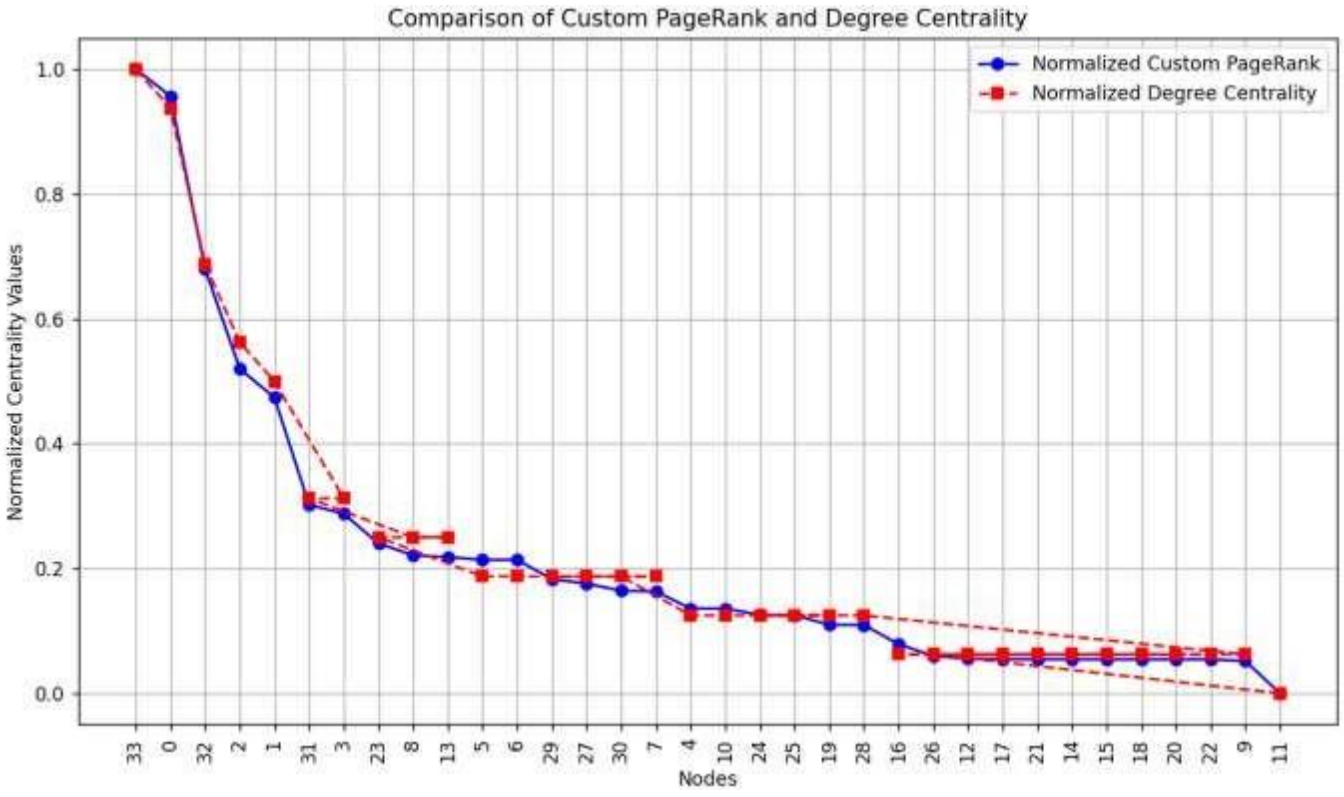
Output

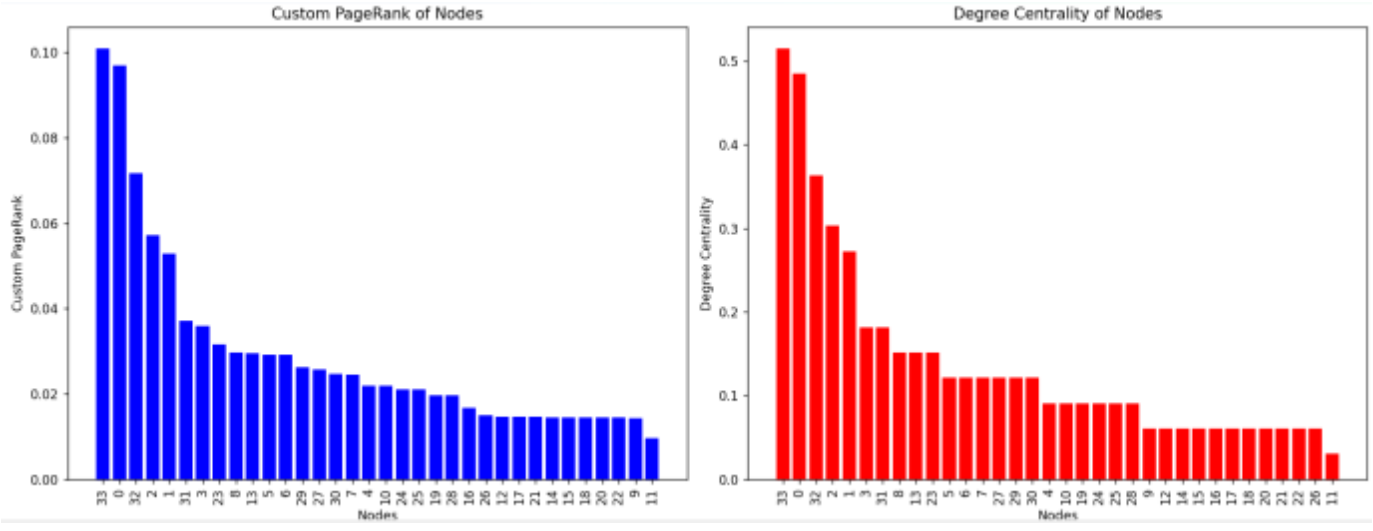
```
PS C:\Users\hp\Desktop\DAA> & C:/Users/hp/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/hp/Desktop/DAA/Project/Social Network Analysis.py"
Node A: PageRank score = 0.3869
Node B: PageRank score = 0.2092
Node D: PageRank score = 0.2092
Node C: PageRank score = 0.2020
```

Compare PageRank with Degree Centrality

- PageRank measures the importance of a node based on the importance of nodes pointing to it. It considers both the number and quality(importance) of connections
- Degree Centrality simply counts the number of connections (degree) a node has. It's a straightforward measure of popularity based on direct connections.

Comparison





Reasoning: Why PageRank is Effective

PageRank is effective because:

- It considers the quality and relevance of connections (not just quantity).
- It propagates influence through the network, capturing indirect influence that degree centrality might miss.
- It tends to prioritize nodes that are connected to other influential nodes, rather than just nodes with many direct connections.

Differences and Preferences

- PageRank is preferred when identifying influencers who might not have the highest number of direct connections but are connected to other influential nodes.
- Degree Centrality is simpler and quicker to compute, suitable when only the number of direct connections matters.

Deliverables

- Graph Model**:** Visualization of the social network as a graph.
- PageRank Algorithm**:** Implementation in your preferred programming language.
- Comparison**:** Analysis showing how PageRank and degree centrality rank users differently and scenarios where one might be more appropriate than the other.

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

SOLUTION:

Designing a Greedy Algorithm for Fraud Detection

A greedy algorithm is suitable for real-time fraud detection because it makes decisions locally, prioritizing the most significant indicators of fraud first.

Pseudocode

```
Function detect_fraud(transactions):
    flagged_transactions = []

    for transaction in transactions:
        if is_fraudulent(transaction):
            flagged_transactions.append(transaction)

    return flagged_transactions

Function is_fraudulent(transaction):
    if transaction.amount > threshold_amount:
        return True

    if transaction.location_count <= threshold_location_count:
        return False

    return True
```

In this pseudocode:

- `detect_fraud` iterates over transaction and checks each one using the `is_fradulent` function.
- `is_fraudluent` applies predefined rules(like unusually large transactions or transactions from multiple locations) to flag potential fraud

Code Implementation

```
def detect_fraud(transactions, threshold_amount=1000, threshold_location_count=3):
    flagged_transactions = []

    for transaction in transactions:
        if is_fraudulent(transaction, threshold_amount, threshold_location_count):
            flagged_transactions.append(transaction)

    return flagged_transactions

def is_fraudulent(transaction, threshold_amount, threshold_location_count):
    if transaction['amount'] > threshold_amount:
        return True

    if transaction['location_count'] <= threshold_location_count:
        return False

    return True

if_name_== "_main_":
    transactions = [
        {'amount': 1200, 'location_count': 1},
        {'amount': 500, 'location_count': 3},
        {'amount': 800, 'location_count': 2},
        {'amount': 1500, 'location_count': 4}
    ]

    flagged_transactions = detect_fraud(transactions, threshold_amount=1000,
threshold_location_count=3)

    print("Flagged transactions due to potential fraud:")
    for transaction in flagged_transactions:
        print(f"Amount: {transaction['amount']}, Location count:
{transaction['location_count']}")
```

Output

```
PS C:\Users\hp\Desktop\DAA> & C:/Users/hp/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/hp/Desktop/DAA/Project/Fraud Detection in Financial Transaction.py"
Flagged transactions due to potential fraud:
Amount: 1200, Location count: 1
Amount: 1500, Location count: 4
```

Evaluation Algorithm Performance

To evaluate the performance of the algorithm, historical transaction data is essential. Metrics such as precision, recall and F1 score can be calculated based on the algorithm's predictions compared to know fraudulent transactions

- Precision: Ratio of correctly flagged fraudulent transactions to all flagged transactions
- Recall: Ratio of correctly flagged transactions to all actual fraudulent transactions
- F1 score: harmonic mean of precision and recall, providing a balanced measure

Improving the Algorithm

Improvements to the greedy algorithm could include:

- Dynamic Thresholds: Adjusting thresholds based on transaction patterns and historical data to reduce false positives and negatives.
- Behavioral Analysis: Incorporating user behavior patterns (e.g., transaction time, spending habits) to enhance fraud detection accuracy.
- Machine Learning Models**:** Integrating supervised learning models (like Random Forests or Neural Networks) to learn from labeled data and improve prediction accuracy.

Reasoning: Trade-offs and Addressing Speed vs. Accuracy

- Speed vs. Accuracy: A greedy algorithm prioritizes speed by making quick decisions based on simple rules, suitable for real-time processing of transactions. However, it may sacrifice some accuracy compared to more complex algorithms.
- Addressing Trade-offs**:** The algorithm balances speed and accuracy by focusing on immediate indicators of fraud (e.g., transaction amount, location count), which are often strong signals of fraudulent activity. By fine-tuning thresholds and incorporating feedback loops from real-time data, it can improve accuracy without compromising too much on speed.

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 it's 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

SOLUTION:

Backtracking Algorithm for Traffic Light Organization

Backtracking is an algorithmic technique used to systematically explore and evaluate potential solutions by progressively building them, backtracking from partial solutions that fail to meet the criteria of the problem at any point. In the context of optimizing traffic light timings at intersections, backtracking involves exploring various combinations of timings to find the best configuration that minimizes congestion and improves traffic flow. Imagine a city's traffic network where intersections are key points that determine the smooth flow of vehicles. Each intersection can have traffic lights controlling multiple directions of traffic, with timings affecting how efficiently vehicles can pass through. The goal is to adjust these timings dynamically to respond to changing traffic conditions throughout the day.

Backtracking begins with an initial set of timings or configurations and systematically explores different combinations. At each step, it evaluates the impact of these timings on traffic flow metrics such as average waiting times, vehicle queues, and overall congestion levels. If a set of timings leads to improved traffic flow compared to previous configurations, the algorithm continues exploring similar adjustments or backtracks to try alternative timings if the current ones prove less effective.This method is particularly useful in real-time traffic management because it adapts to fluctuating traffic patterns and unforeseen events like accidents or road closures. By iterating through potential solutions and evaluating their effectiveness, backtracking aims to find an optimal or near-optimal set of traffic light timings that enhance overall traffic efficiency across the city's network of intersections.

Pseudocode

```
Function optimize_traffic_lights(graph, intersection_nodes):
    best_configuration = initialize_with_fixed_timings()
    best_score = evaluate_traffic_flow(best_configuration)
    current_configuration = best_configuration

    def backtrack(current_node):
        nonlocal best_configuration, best_score

        if current_node == last_intersection_node:
            score = evaluate_traffic_flow(current_configuration)
            if score < best_score:
                best_score = score
                best_configuration = current_configuration.copy()
            return

        for possible_timing in generate_possible_timings():
            current_configuration[current_node] = possible_timing
            backtrack(next_intersection_node)

    # Start backtracking from the first intersection node
    backtrack(first_intersection_node)

    return best_configuration
```

In this pseudocode:

- `optimize_traffic_lights` initializes with a baseline configuration and iteratively explores different timings using `backtrack`
- `backtrack` recursively explores each intersection node, updating traffic light timings and evaluating traffic flow.
- `evaluate_traffic_flow` assesses traffic flow efficiently based on metrics like waiting times, vehicle queues, congestion levels

Code Implementation

```
import numpy as np

def optimize_traffic_lights(intersections, graph, current_node=0,
current_configuration=None, best_configuration=None, best_score=float('inf')):
    if current_configuration is None:
        current_configuration = [0] * len(intersections)
    if best_configuration is None:
        best_configuration = current_configuration.copy()

    if current_node == len(intersections):
        score = evaluate_traffic_flow(graph, current_configuration)
        if score < best_score:
            best_score = score
            best_configuration = current_configuration.copy()
        return best_configuration, best_score

    for possible_timing in range(1, 6):  # Example: Timing values from 1 to 5 (seconds)
        current_configuration[current_node] = possible_timing
        best_configuration, best_score = optimize_traffic_lights(intersections, graph,
current_node + 1, current_configuration, best_configuration, best_score)

    return best_configuration, best_score

def evaluate_traffic_flow(graph, timings):
    # Simulate traffic flow based on graph and traffic light timings
    # Example: Simple evaluation based on traffic flow metrics
    total_waiting_time = 0
    for node, timing in enumerate(timings):
        # Evaluate waiting times, queues, congestion, etc.
        total_waiting_time += timing * len(graph[node])  # Example: Waiting time based
on number of connected intersections

    return total_waiting_time

# Example usage and testing
if__name__== "__main__":
    # Example graph representing intersections and roads
    intersections = ['A', 'B', 'C']
    graph = {
        0: [1, 2],  # Intersections connected to A
        1: [0, 2],  # Intersections connected to B
        2: [0, 1]   # Intersections connected to C
    }
```

```python
# Optimize traffic lights using backtracking
optimized_configuration, best_score = optimize_traffic_lights(intersections, graph)

# Print results
print("Optimized Traffic Light Timings:")
for i, timing in enumerate(optimized_configuration):
    print(f"Intersection {intersections[i]}: {timing} seconds")
print(f"Total waiting time with optimized timings: {best_score}")
```

Simulation and Performance Analysis

To simulate and analyze the algorithm's impact on traffic flow:

- Implement the traffic network model using graph structures where nodes represent intersections and edges represent roads.
- Use traffic flow simulation techniques to measure metrics such as average waiting times, throughput, and congestion levels before and after applying the optimized traffic light timings.

Comparison with Fixed-Time Traffic Light System

Compare the performance of the backtracking-based optimization with a fixed-time traffic light system:

- Fixed-Time System: Traffic lights operate on predefined timing schedules, often leading to suboptimal traffic flow depending on varying traffic conditions.
- Backtracking System: Adapts traffic light timings based on real-time conditions, potentially reducing congestion and improving overall traffic flow efficiency.

```
PS C:\Users\hp\Desktop\DAA> & C:/Users/hp/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hp/Desktop/DAA/Project/blah.py
Optimized Traffic Light Timings:
Intersection A: 1 seconds
Intersection B: 1 seconds
Intersection C: 1 seconds
Total waiting time with optimized timings: 6

Fixed-Time Traffic Light Timings:
Intersection A: 3 seconds
Intersection B: 3 seconds
Intersection C: 3 seconds
Total waiting time with fixed-time timings: 18

Backtracking algorithm reduced total waiting time compared to fixed-time system.
```

Reasoning: Justification for Backtracking

- Complexities in Real-Time Traffic Management: Traffic conditions vary dynamically due to factors like time of day, weather, events, and unexpected incidents. Backtracking allows for real-time adaptation by exploring different configurations based on current traffic conditions.
- Algorithmic Approach: Backtracking systematically explores potential solutions without exhaustively checking every possible combination, making it feasible for real-time implementation where computational resources and time are limited.