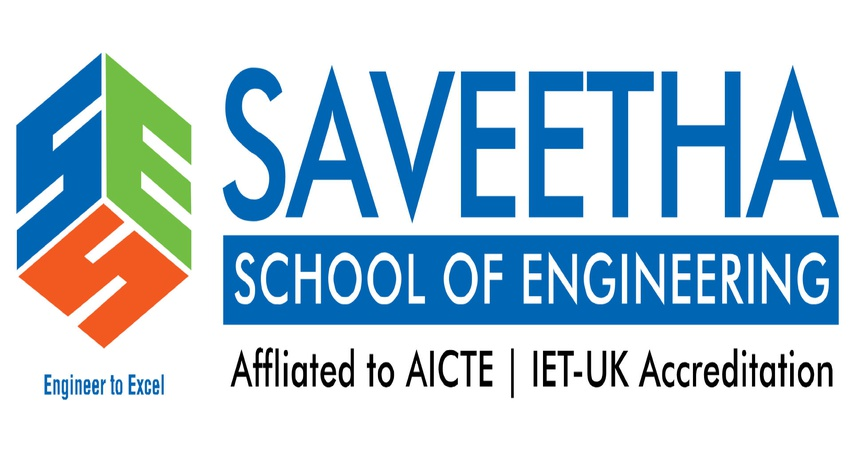
****

**Assignment**

**SAVEETHA SCHOOL OF ENGINEERING**



Submitted by

**PHILIP GRACIAN SANOVA S (192324023)**

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

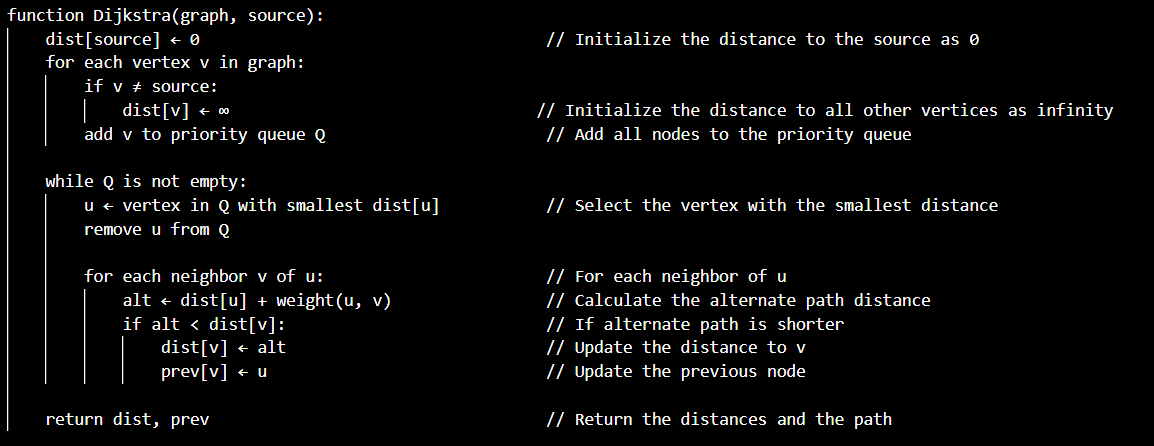
Graph Representation

* **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:

****

Code Implementation:

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}

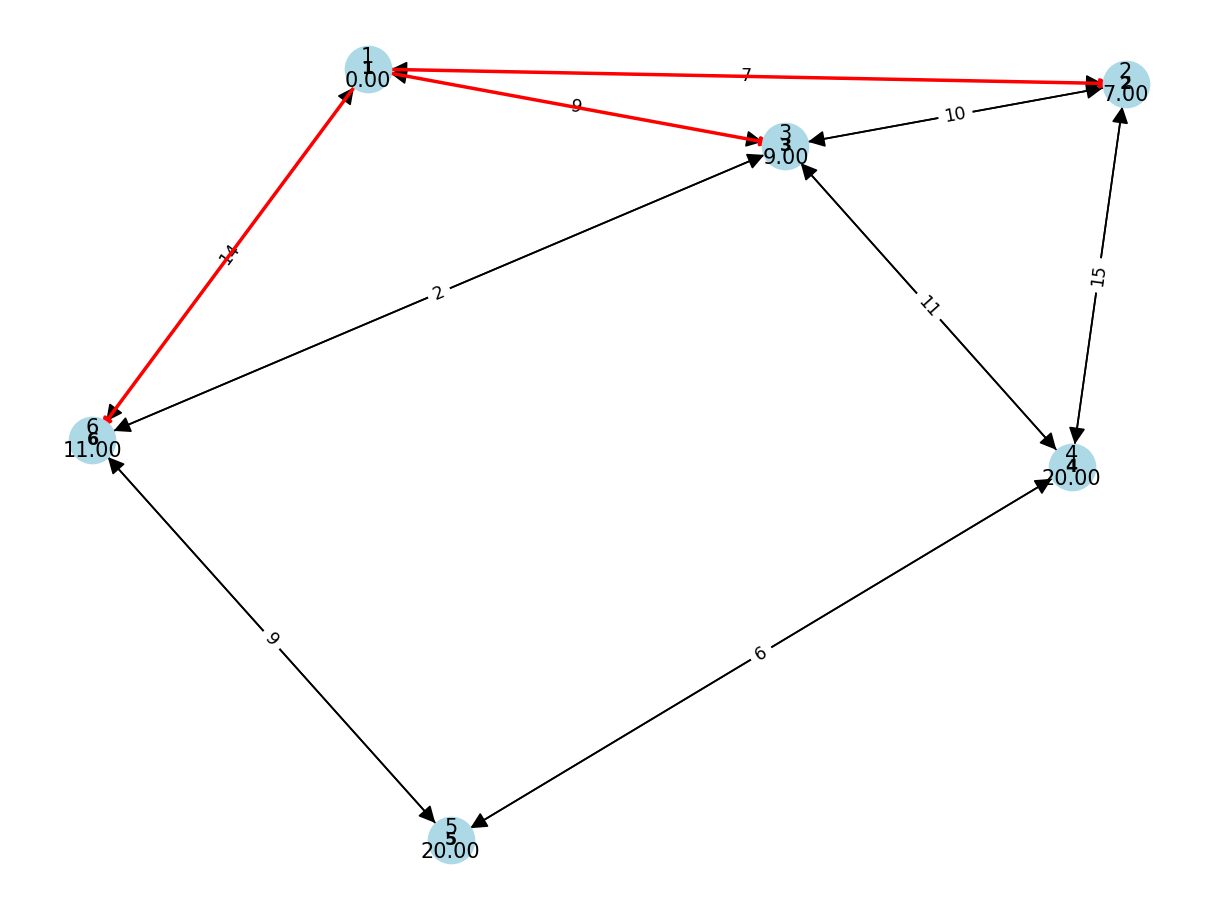
nx.draw\_networkx\_labels(G, pos, labels=node\_labels, font\_color='black')

# 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: ,where V is the number of vertices and E is the number of edges.
* Space Complexity: 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:**

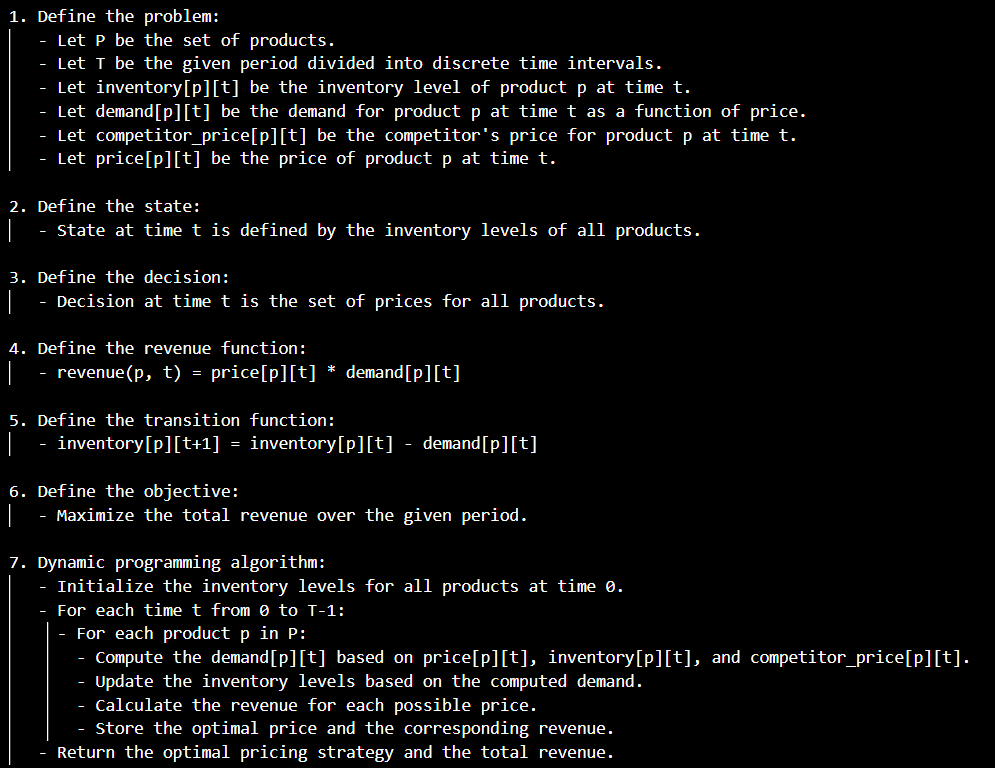
* Pseudocode and implementation of the dynamic pricing algorithm
* Simulation results comparing dynamic and static pricing strategies
* Analysis of the benefits and drawbacks of dynamic pricing

**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

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

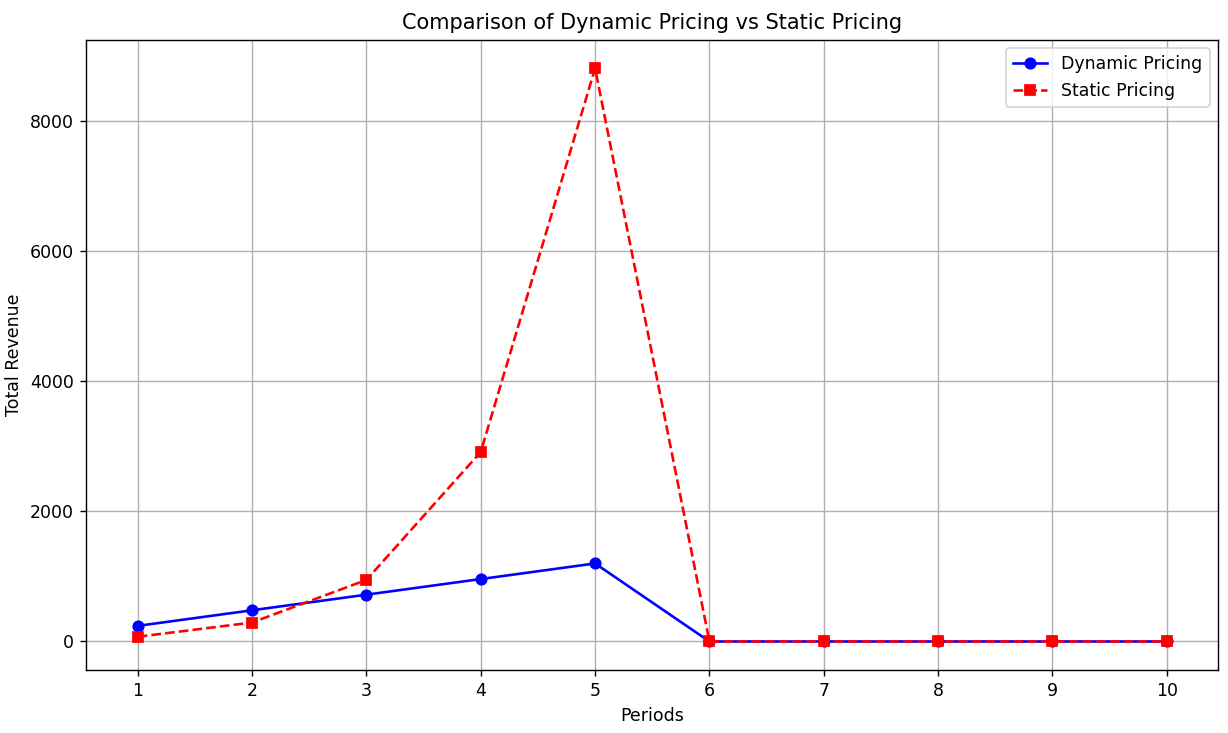


### Analysis of the Benefits and Drawbacks of Dynamic Pricing

#### 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:**

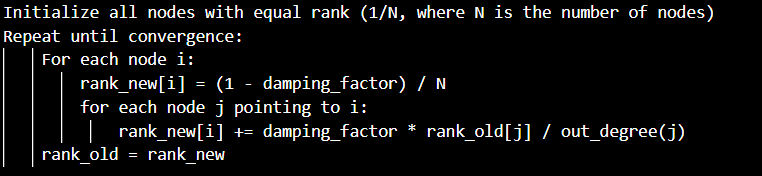
**Model the Social Network as a Graph**

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



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']

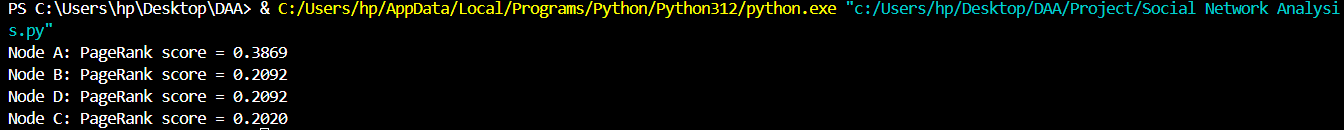
    }

    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}")

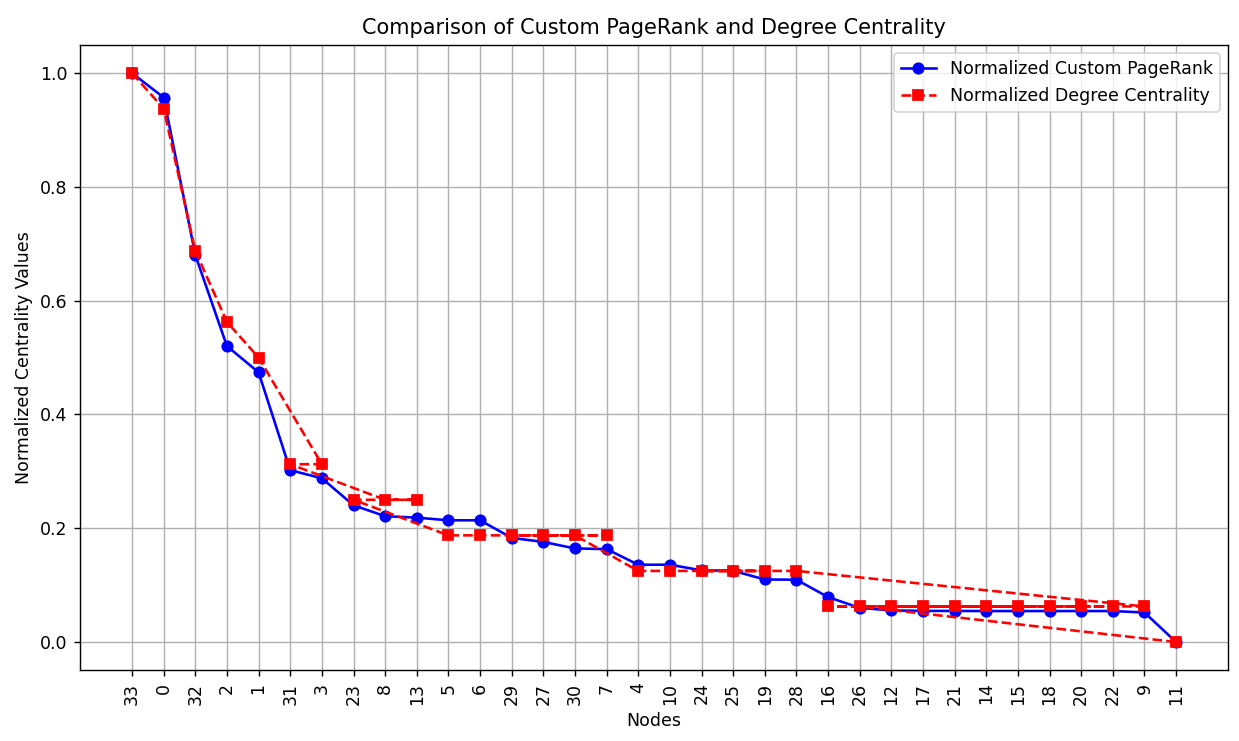
Output

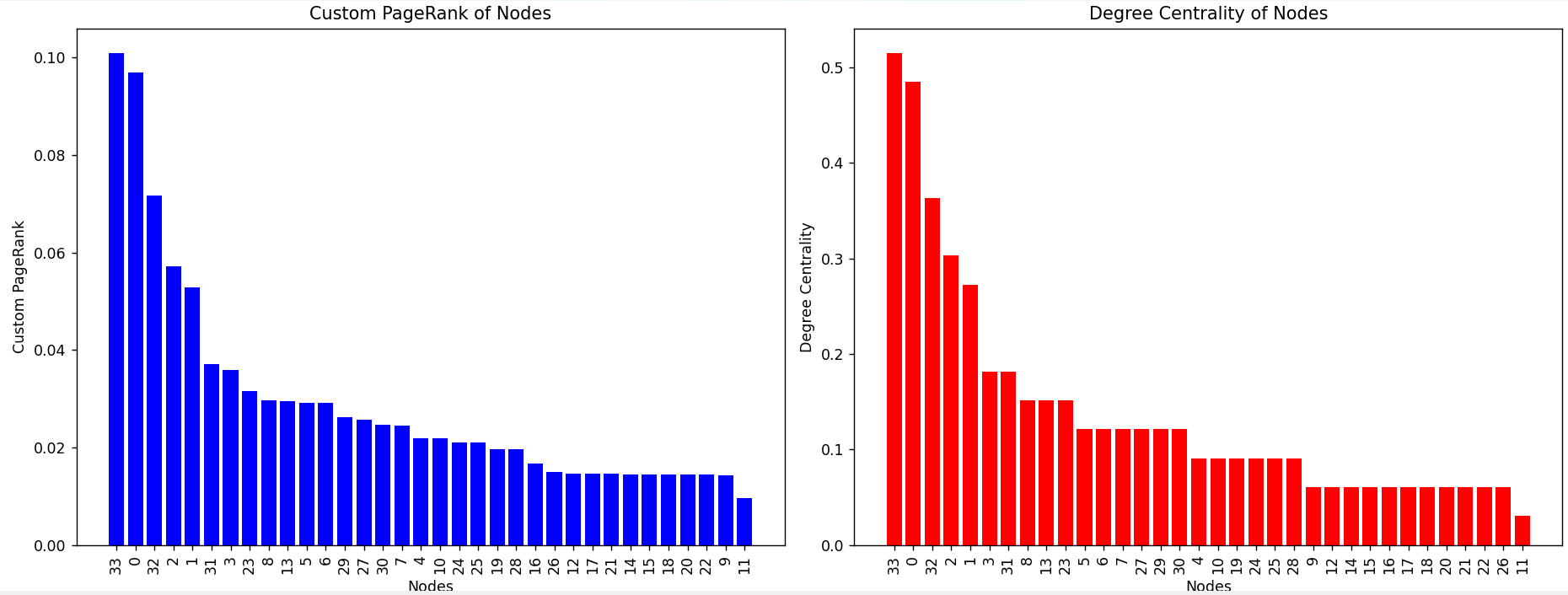


**Compare PageRank with Degree Centrality**

* 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

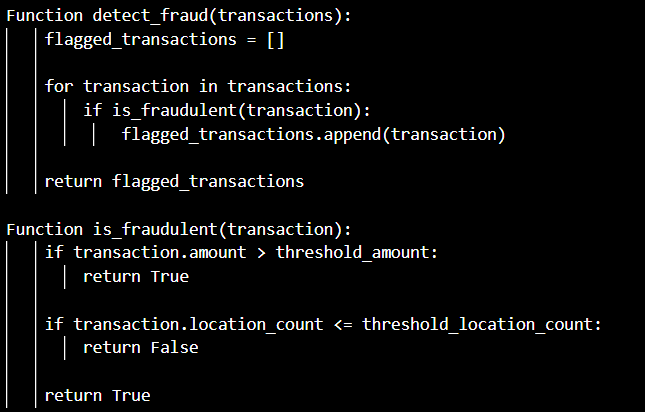
**Reasoning:** Explain why a greedy algorithm is suitable for real-time fraud detection. Discuss the trade-offs between speed and accuracy and how your algorithm addresses them

**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



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}

    ]

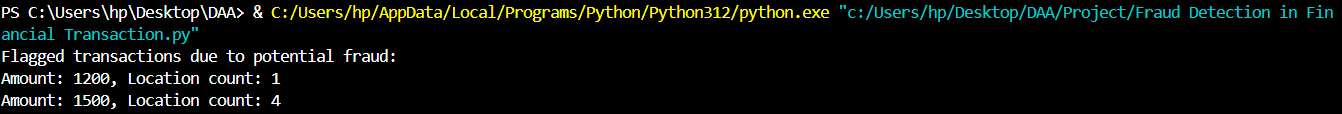
    flagged\_transactions = detect\_fraud(transactions, threshold\_amount=1000, threshold\_location\_count=3)

    print("Flagged transactions due to potential fraud:")

    for transaction in flagged\_transactions:

        print(f"Amount: {transaction['amount']}, Location count: {transaction['location\_count']}")

Output



**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

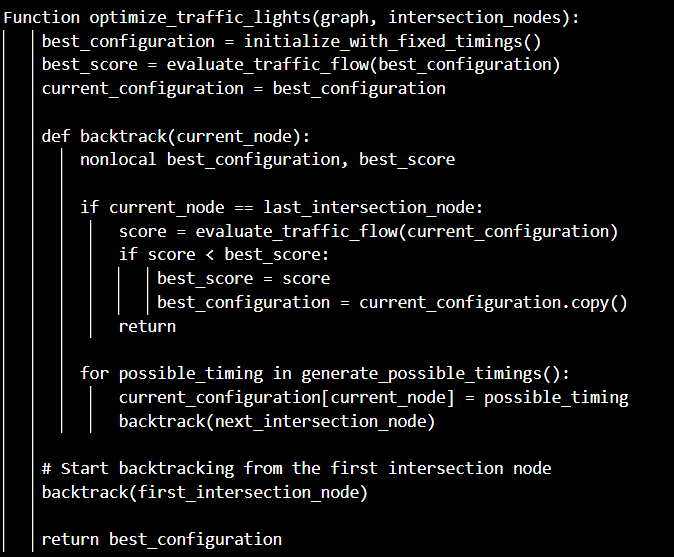
**Reasoning:** Justify the use of backtracking for this problem. Discuss the complexities involved in real-time traffic management and how your algorithm addresses them

**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 

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

    }

    # 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

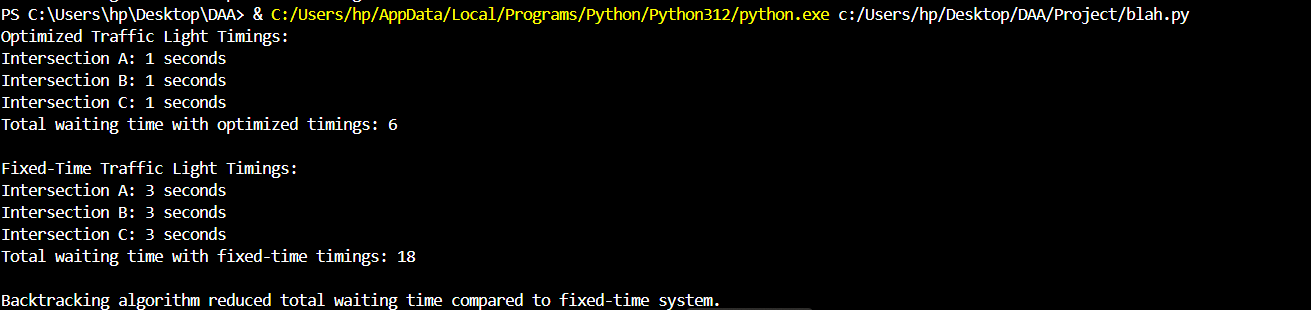
To simulate and analyze the algorithm's impact on traffic flow:

* 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.



### 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.