Date:- 1/07/24

CSA-0676 DAA

K.Vamsi
192372062

Assignment-

Problem 1:-

Optimizing Delivery routes:

Modelling the city's Road network as a graph.

To model the city's road network as a graph we can represent each intersection as a node and each road connecting two intersections as an edge. The weights of edges can represent the travel time between the connected intersections.

Here's an example of how the graph could be represented.

graph :- {

'A' = {'B':5, 'C':1},

'B' = {'A':5, 'C':2, 'D':1},

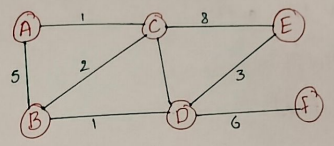
'C' = {'A':1, 'B':2, 'D':4, 'E':8},

'D' = {'B':1, 'C':4, 'E':3, 'F':6},

'E' = {'C':8, 'D':3},

'F' = {'D':6}

The above example graph should be represented as.



In this example. the graph is represented as a dictionary each key represents a node and its value is another dictionary containing the neighboring nodes and their corresponding weights.

Implementing Dijkstra's Algorithm:-

Dijkstra's Algorithm is suitable for this problem because it find the shortest Path between a source node and all other nodes in a weighted graph with non-negative edges weight. Here the Pseudo code for Dijkstra's Algorithm

Pseudo codes

```
function Dijkstra (graph, source):
    for each vertex v in graph:
        dist [v] = Infinity
        Prev [v] = undefined

    dist [source] = 0
    Q := the set of all nodes in graph
    while Q is not empty:
        u := vertex in a Q with smallest dist [u]
    remove u from Q
    for each neighbour v of u still in Q:
        alt := dist [u] + weight (u,v)
        if alt < dist [v]:
            dist [v] := alt
            Prev[v] := u

    return dist, Prev
```

```
import heap q
def dijkstra (graph, source):
    dist = {node: float ('inf') for node in graph}
    dist [source] = 0
    Prev = {node: None for node in graph}
    heap = [(0, source)]
    while heap:
        current_dist, current_node = heap q. heap pop (heap)
        if current_dist > dist [current_node]:
            continue
    for neighbour, weight in graph [current_node]. items():
        distance = current_dist + weight.
        if distance < dist [neighbour]:
            dist [neighbour] = distance
            Prev [neighbor] = current_node
        heap.q. heap push(heap, (distance, neighbour))
    return dist, Prev.
```

This implementation users a min heap efficiency

Analyzing the efficiency and potential improvements.

Dijkstras Algorithm has a time complexity of $O((|V| + |E|) \log |V|)$ using a binary heap (∞) $O(|V| + |E|) \log |E|)$ using a fibonacci heap where $|V|$ is the number of vertices (intersections) and $|E|$ is the number of edges (roads). The space complexity is $O(|V|)$ for storing the distance and previous nodes

Potential Improvements:

1. Use a more efficient data structure:
   Instead of a binary heap. using a fibonacci heap can improve the time complexity to $O((|V| + |E|) \log |V|)$

2. Implement bidirectional search:-
   By searching from both the source and destination simultaneously, the search space can be reduced Potentially leading to faster

3. utilize real-time traffic data:
   Incorporating live traffic data into the edge weighs can provide more accurate and up-to data route recommendation.

Assumptions and considerations

1. Non-negative weights:
   Dijkstra's Algorithm assume that all edges weight (travel times) are non-negative. If negative weight are present, the algorithm may not find the correct shortest path.

2. Static road networks:
   The current implementation assumes a static road network. In reality, road conditions can change due to factors such as traffic road closures (∞) constructions.

(25) Problem:- 4

FRAUD Detection in finacial transactions

1 Greedy algorithm for fraud detection

To detect potentialy fraudlent transaction in real time a greedy algorithm is well suited for the task. Greedy algorithm makes locally optimal choices at each stages which can lead globally optimal solution for problem that exhibit optimal substructure. In the context of fraud detection this means that the algorithm can quickly flag suspicious transaction based on a set of predefined rules, without the needs for complex computation (or) extensive data analysis.

Pseudo code:

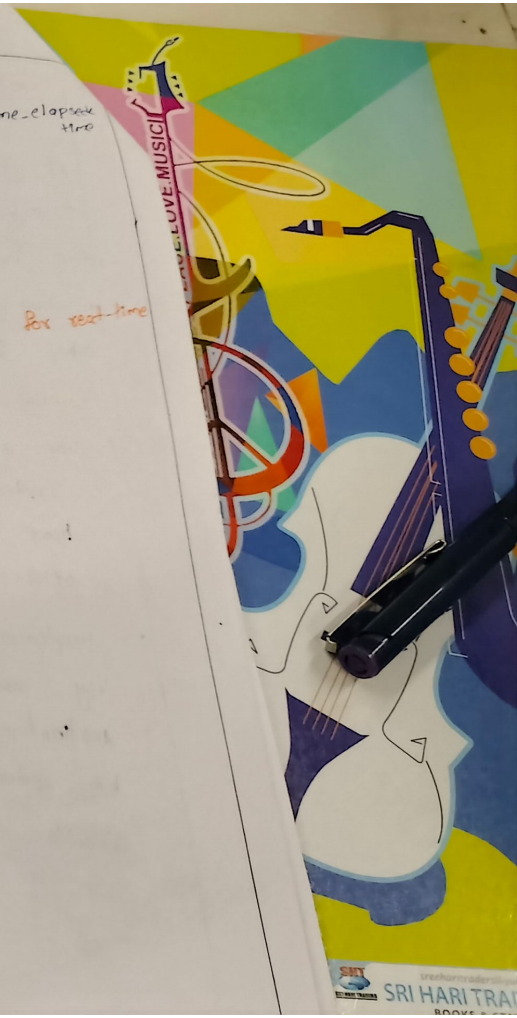
```
function detect fraud (transaction):
    rules = [
        "unusually-Large-transaction",
        "multiple-location-in-short-time",
        "unusal-transaction-pattern
    for rule in rules:
        if violates rule (transaction, rule):
            return true

function violates rule (transaction, rule):
    return True
    return transaction amount > threshold
elif rule == "multiple-location-in-short-time":
    return transaction locations count() > 1 and transaction time-elapsed time
elif rule == "unusal-transaction-pattern":
    return false
else
    return false.
```

The key advantages of using a greedy algorithm for real-time fraud detection are:

1. speed
2. simplicity
3. flexibility.

Program for fraud detection in financial transaction.

```
def evaluate-performance(transactions)
    true-positives = 0
    false-positives = 0
    true-negatives = 0
    false-negatives = 0
    for transaction in transactions:
        if detect fraud (transaction):
            if transaction-is-fraud length:
                true-positive += 1
        else:
            false-positive += 1
```



```
else:
    if transaction is fraudlent:
        false_negatives += 1
    else
        true_negative += 1

Precision = true_Positive / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
return precision, recall, f1_score
```

2. Performance evaluation

1. Precision:-
The proportion of transaction flagged as fraudlent that are actually fradient.

2. Recall:-
The proportion of actual fradient transaction that are cossectly identified.

3. F1-score:-
The hosmonic mean of precision and recall providing a balaneed measure of the algorithm performance

3. Improvements to the Algorithm

1. Incorporate machine learning:-
Instead of relying solely on predefined rules, we can leverage machine learning technique to build a more suphisticated fraud detection model this can involve training a classifier on historical transaction data to learn of

2. Adaptive thresholds:-
Instead of using fixed thresholds for the rules (eg. $10,000 for unusually large transactions). we can make the threshold adaptive based on the users spending pattern (or) other contextual information. This can help reduce the number of false negatives.

3. Ensemble Technique:-
Combining multiple fraud detection algorithm (eg. the greedy algorithm and a machine learning based model) can help the algorithm to improve overall performance by leveraging the strength of each approach.

4. Continuous learning:-
Regularly updating the fraud detection model with new transaction data can help the algorithm adapt to evolving fraud pattern and maintain high performance over time.

5. Explainability:-
Providing explanations for the fraud detection decision can help build trust and transparency which is crucial in the financial industry.

(8) Problem - 5:

Real time traffic management system:

Back tracking algorithm for traffic light optimization:

1. Dynamic optimization:

Backtracking allows the algorithm to explore different possible configuration of traffic light timings and select the optimal solution based on the current traffic conditions. This is crucial in a real time system where traffic patterns can change rapidly.

2. Constraint Handling:-

Backtracking can effectively handle the various constraints involved in traffic light optimization such as ensuring smooth traffic flow minimizing waiting times and balancing the needs of different traffic streams.
(Eg: vehicles, Pedestrains, Public transportation).

Pseudo codes

```
function optimize Traffic lights (traffic networks):
    intial state = getInitial (traffic network)
    return backtrack (initial state, traffic network)
function backtrack (current state, traffic network):
    if is goal state (current state, traffic network):
        return current state
    if result is none:
    return result
return none
function get next states (current state, traffic network):
    next states = [ ]
for intersection in traffic network. intersection:
    for new timings in get new timing (current state, intersection):
next state = update state (current state, intersection, new timings)
    next state . append (next state)
    return next states
function is valid (state, traffic network):
    return True.
function is goal state (state, traffic network);
    return True.
from typing import list, Tople
class traffic light:
    def-int (self, intersection_id : int , green_duration : int, yellow_duration : int)
    self.intersection_id = intersection_id
    self .green_duration = green_duration
    self yellow_duration = yellow duration.
```

2. Simulation and Performance Analysis:-

1. Traffic network model:

A detailed representation of the city's road network, including the locations and configurations of intersections, traffic lights, and other relevant infrastructure.

2. Traffic demand model:-

A model that generates realistic traffic demand patterns, taking into account factors such as time of day, day of week, and special event.

3. Traffic flow model:-

A model that accurately simulates the movement of vehicals, pedestrains and other road users, considering factors such as vehicles dynamic, driver behaviour, and traffic signal timing.

4. Performance metrics.

Relevant metrics to evaluate the performance of the traffic light optimization algorithm, such as average vehicle delay, queue length and comparison with fixed-time, Traffic-light system.

1. Responsiveness to traffic conditions:-

The backtracking algorithm should demonstrate superior performance in adapting to changing traffic conditions as it can dynamically adjust the traffic l timing whereas the fixed-time system is limited in it limited in its ability to respond to real time changes.

2. Scalability and flexibility:-

The backtracking algorithm should be able to handle large and more complex traffic networks, with the ability t scale and adapt to the changing needs of the city. whereas the fixed-time system mass have limitation in ability.

3. Computational complexity:-

while the backtracking algorithm may require mo computational resources than the fixed time system the improved performance and adaptability should justif the additional computational cost. especially in the context of real time traffic management.

4. Problem - 28-

Dynamic Pricing algorithm for E commerce

1. Dynamic Programming algorithm for optimal pricing Strategy.

The dynamic Pricing Problem for an ecommerce company can be effectively solved using a dynamic Programming Approach. Dynamic Programming is well-suited for this problem because it allows us to break down the complex decision-making process into smaller, interconnected subproblems and solve them efficiently.

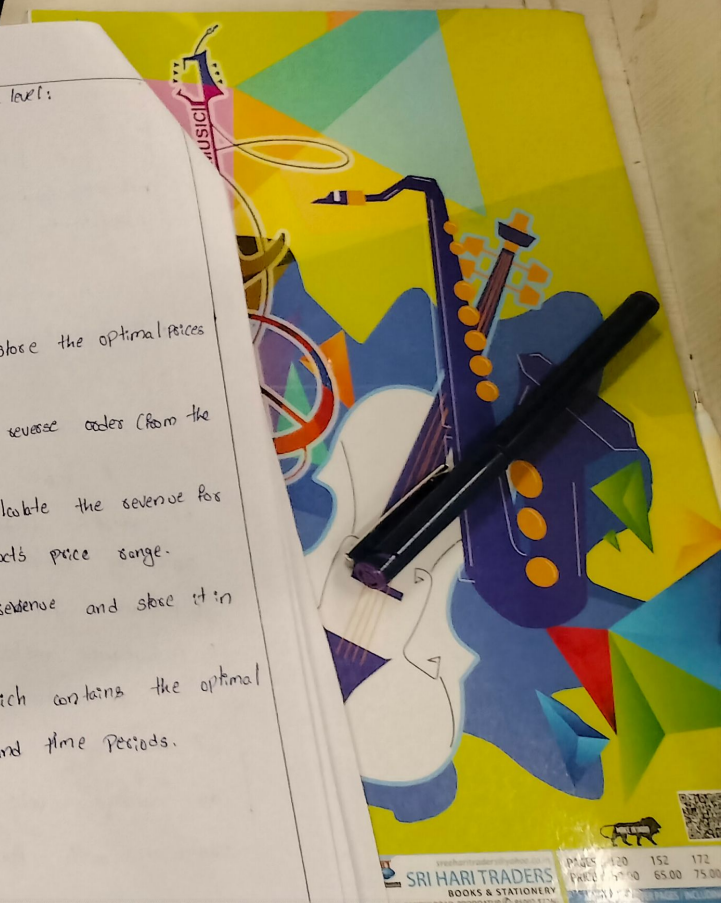
Pseudo code:-

```
function optimal_Pricing_strategy (Products, time_Periods):
    optimal_Prices = [[[ 0 for k in range (len(Products))]
                        for j in range (time_Periods)]
                        for i in range (time_Periods)]

    for t in range (time_Periods-1, -1, -1):
    for P in range (len(Products)):
        max_revenue = 0
    for price in range (Products[P].min_Prices.
        Products[P].demand (price+1);
    revenue = price * Products[P].demand (price, 1, competitions.
```

```
def dynamic_Pricing_algorithm (base_price, demand_level):
    if demand_level == "high":
        return base_price * 1.5
    elif demand_level == "medium":
        return base_price * 1.2
    else return base price * 1.0
```

The key steps of the algorithm are

1. Initialize a 3D array optimal_Prices to store the optimal prices of each Product and time Period.
2. Iterate through the time Periods in reverse order (from the last Period to the first).
3. For each Product and time Period, Calculate the revenue for all possible prices, within the Product's price range.
4. Find the Price that maximize the revenue and store it in the 'optimal_Prices' array.
5. Return the optimal_Prices array which contains the optimal Pricing strategy for all Products and time Periods.



The algorithm takes into account the following factors.

* Inventory levels
* competitor pricing
* Demand elasticity

2. simulation and comparision with static pricing simulation setup.

1. Generate a set of products with varying price range demand function and competitor prices.
2. Implement the dynamic pricing algorithm to determine the optimal pricing strategy for the products over a given time period.
3. Run both pricing strategies and compare the total revenue generated.

3. Analysis of Benefits and Drawbacks:

Benefits of Dynamic Pricing.

1. Increased Revenue:

The dynamic pricing algorithm can adjust prices in real time to maximize revenue based on demand and competitor prices

* Implement a simple static pricing strategy where the prices are fixed throughout the time period.

2. Improved inventory management

The algorithm can consider inventory levels when determining optimal prices helping the avoid stockouts (or) excess inventory

Drawback of dynamic pricing.

* complexity:

Implementing a dynamic pricing algorithm can be more complex than a static pricing strategy, requiring more data computational resources, and ongoing maintenance.

2. Data Requirements.

The algorithm relies on accurate data about demand competitor prices and other factors can be challenging to obtain and maintain

⑤ Problem-3:

Social Network Analysis:

- Modelling the social network as a graph.

To model the social network as a graph, we can represent users as nodes and their connection as edges. Each node would represent a user and an edge b/w two nodes would indicate a connection or relationship b/w the users.

The graph can be represented as adjacency matrix (or) an adjacency list, depending on the size and sparsity of the network. The adjacency matrix would be a square matrix where the element at row 'i' and column 'j' represents the weight of edge between 'i' and 'j'. The adjacency list would be a collection of lists where each list contains the neighbours of particular node.

- Implementing the PageRank algorithm.

The PageRank algorithm is an effective way to identify influential users in a social network. The algorithm assigns a numerical score to each node (or) user in the network, based on the importance (or) influence of that node within the network.

Pseudo code:

```
function pageRank(G, d, max_iterations, epsilon):
    n = number of nodes in the graph G
    PR = [1/n]*n
    for i in range(max_iterations):
        new_PR = [0]*n
        for u in range(n):
            for v in G.neighbors(u):
                new_PR[u] += d*PR[v]/len(G.neighbors(v))
            new_PR[u] += (1-d)/n
        if sum(abs(new_PR[j] - PR[j]) for j in range(n)) < epsilon:
            break
        PR = new_PR
    return PR
```

```
import networkx as nx
import matplotlib.pyplot as plt.
G = nx.karate_club_graph()
nx.draw(G, with_labels = True)
plt.show()
```

3. Comparing PageRank and Degree centrality.

The key difference between Pagerank and degree centrality are.

1. Consideration of Network structure:

PageRank considers the entire network structure including the importance of the nodes a user is connected to, while degree centrality only considers the number of connections a user has.

2. Recursive Influence.

PageRank assigns higher scores to user who are connected to other Influential users. whereas degree centrality treats all connections equally.

3. Convergence and stability

The PageRank algorithm converages to a stable set of scores while degree centrality can be more sensitive to changes in the network.

Reasoning:

1. PageRank is an effective measure for identifying because influential users in a social network because it considers the overall structure and importance of the network. rather than just the number of connections a user has by taking into account the importance of nodes a user is connected to PageRank can identify users who may have fewer connections but are connected to other influential users and thus have a greater impact on the networks

2. The recursive nature of the PageRank algorithm where the score of a node is influenced by the score of its neighbours allows it to capture the complex relationships and flow of information within the social networks. This makes PageRank a more Robust and reliable measurable of influence compared to simple degree centrality.