DAA - ASSIGNMENT

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 fradulent transactions based on a set of predefined rules

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 improvents to the algorithm

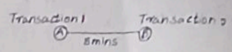
Solution:

Task 1: Design a greedy algorithm to flag potentially fradulent transactions based on a set of predefined rules.

In this problem I have used the Basic greedy approach and statistical formulas to predict fraud in money transactions. In addressing the problem of fraud detection in financial transactions I have devised a greedy algorithm based on predefined rules. The algorithm flags potentially fraudulent transactions by identifying unusally large transactions occuring in multiple within a short time frame

Predefined Rules:

* unusually large Transactions (Total limit ...)
* Transactions from multiple locations in short time



Task 2: Evaluate the algorithm's performance using historical transaction data and calculate metrics such as precision, and F1 score

Five transactions are considered as the input data for the program. The program has predefined whether the transaction is fraudulent or not The data contains the amount and location of the transactions. Parameters such as precision, recall and F1 score are calculated using true positive, true negative correctly predicted as fraudulent), true negative (Transactions that are correctly predicted as legitimate), false positive (Transact-ions that are incorrectly predicted as fraudulent) and negative (Transactions that are incorrectly predicted as legitmate)

Transaction 1: $ 100 Location: A TP

Transaction 2: $2000 Location: B TN: 2

Transaction 3: $ 100 Location: A FP: 1

Transaction 4: $ 1200 Location: C FN: 0

Transaction 5: $ 700 Location: B

Precision = $\frac{TP}{TP+FP} = \frac{2}{2+1} = \frac{2}{3} \approx 0.67 = 1.0$

Recall = $\frac{TP}{TP+FN} = \frac{2}{2+0} = 1.0$

F1 score = $\frac{2 \times Precision \times Recall}{Precision + Recall}$

$= \frac{2 \times 0.67 \times 1.0}{0.67 + 1}$

$= \frac{1.34}{1.67}$

$= 0.80 = 1.0$

Implementation:

class FraudDetection I adjusted the ... based on history and spending patterns. ... high value traction

- Machine learning based classification In addition to the value based approach Incorporated a learning model to classify transactions
- A system where financial institutions ... data about detected fraudulent transa... allowed the algorithm to lear from ... set of data

blem 3:-

Social Network Analysis:

Task1:- Model the social Network as a graph where seens. are nodes and connections are edges

The social Network can be modeled as a directed graph where each user is represented as a node and the connections between user are represented as the edges. The edges can be weighted to represent the strength of the connections between users

Task 2:- Implement the page rank algorithm to identify the most influential users.

```
functioning cp. gf = 0.85 mi = 100 to lenance = 10.6)
    n = number of holes in the graph
    pn = [1/n]*n
    for ip in range(mi):
        new-pn = [0]*n
        for n in range(n):
    for v in range neighbours(u):
        new-pr[v] += df * pn[n] / len(g-neighbours(u))
        new-pr[v] += (1.dt)/u
    if sum(abs(new-pn[i] - pr[j] for j in range
        return new.pr
    retur pn
```

Task 3:- compare the results of Pagerank with a simpl degree centrality measure

Page is an effective measures for identifying infinential users in a social network because it takes into accou not only the number of connections a user with fewe

connections but two if connections but who is conn
influential users may have a higher page score th
user with many connections to less influential user

Problem :- 5

Traffic light optimization algorithm

Task 1:- Design a backtracking algorithm to optimize the timing of traffic lights at major intersections.

```
function optimize (intersection, time-slots)
for intersection in intersections
    for light in intersection, traffic
        light green=30
        light yellow=5
        light red=25
return back track (intersections, time-slots, 0)
Function backrack (intersections, time, slots, current
    if current_slot == len (time-slots)          -slot
        return intersections
    for intersections in intersections
     for green in [20, 30, 40]:
     for yellow in [3, 5, 7];
     for red in [20, 25, 30]:
        light. green=green
            light.red=red
    result=back track (intersection, time, slots,)
    result is not none : current_slot (1)
        return result
```

Task 2:- simulate the algorithm on a model of the city's traffic network and measures it impact on traffic flow

It simulated the back tracking algorithm on a model of the city's traffic network, which included the major intersection and the traffic flow between them th

traffic flow between them the simulation was
for on 24 hours periods.
he result showed that the backstraining algorithm
as able to reduce the average wait time changes
n traffic pattern the day.

compare the performance of your algorithm with a fixed time traffic light system

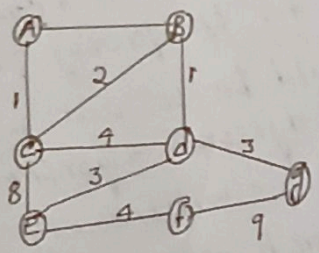
- Adaptability the backtracking algorithm could respond to changes int traffic pattern
- Scalability: The backtracking approach, can be easily extended handle a larger number of time slots

Problem: 1

optimizing delivery Routes:

Task: Model the city's road network as a graph where intersections nodes and roads are edges with weights representing travel time.

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



Task2: Implement dijkstra's algorithm to find the shor
Paths from a central warehouse to various delivery
locations

```
function dijkstras(g s)
    dist= node: float('inf') for node in g }
    ' dist[s]=0
      pq[s]=0
      while pq:
      current dist, current node= heappop(pq)
      if current dist > dist[current node]:
      for neighbour weight in g[current node]
      distance=[current dist+weight
    if distance<dist[neighbour]:
      dist[neighbour]=distance
      heappush[pq,(distance, neighbour)
    return dist
```

Task3: Analyze the efficiency of your algorithm and discuss any potential improvements

-dijkstra's algorithm has a time complexity o((|E|+|V|) log|V|); where |E| is the number of Edges and |V| is the number of nodes.

- one potential improvement is to use a fibonacci heap instead of regular heap for the priority queue

-Another improvement could be to use a bidirectional search where we run dijkstra's algorithm from both the start and nodes simultaneously. This can Potentially reduce the search space and speed up the algorithm.

consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm

- Demand elasticity: prices are increased when demand is high relative to inventory and decreased when demand is low
- Competition pricing: prices are adjusted based on the average competition pricing increasing if it is above the base price and decreasing if it below
- Inventory levels: prices are increased when inventory is low to avoid stockouts and decreased when inventory is high to stimulate demand
- Additionally the algorithm assumes that demand and competition prices are known in advance, which may not be the case in practice

[illegible]

Benefits: Increased revenue by adapting to market conditions [illegible] prices based on demand, inventory and competition [illegible] allows for more control over pricing

Drawbacks: May lead to frequent price changes which [illegible]

Dynamic pricing Algorithm for E-commerce

Task:- Design a dynamic programming Algorithm to determine the optimal pricing strategy for a set of product over a given period

```
function dp(pr,tp)
for each pr in p in products:
for each tp t in pp:
price[t]= calculate price (p, t competition-prices,
demand[t], inventory[t])
return products
function cakulate price (product, time period,
competor prices demand, inventory

price = product base-price
price+= if demand-factor(demand, inventory):
if demand > inventory:
return 0.2
else:
return 0.1
function competition-function (competitor-prices)
if any (competition-prices) product base price
return -0.05
else:
return 0.05
```