Assignment

Name : S. Venkata Praveen.

Subject Code : CSA0676

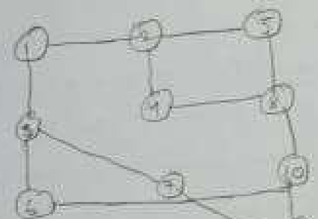
~~Course~~ & Subject : Design analysis of Algorithm

PROBLEM - 3

Social network Analysis

Task 1 :- Model the social network as a graph where users 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 users are represented as 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.

function PR (g - dif = 0.85 , mi = 100 , tolerance = 1e-6)

n = number of nodes in graph

pr = [1/n] * n

for i in range(mi):

new-pr = [0] * n

for n in range(N):

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-slot)
    for intersection in intersection
        for light in intersection.traffic
            light_green = 30
            light.yellow = 5
            light.red = 25
    return back track (intersection, time-slots 0):
function backtrack (intersection, time slot, current slot):
    if current-slot == len(time-slots):
        return intersection
    for intersection in intersection
        for light in intersection_traffic
            for green in [20, 30, 40]:
                for yellow in [3, 5, 7]:
                    for red in [20, 25, 30]
                        light.green = green
                        light yellow = yellow
                        light.red = red.
            result = backtrack (intersection, time slot)
            if result is not None:
                return result.
```

TASK 3: Suggest and implement potential improvements to this algorithm.

→ Adaptive rule thresholds : Instead of using fixed thresholds for rules like unusually large transactions, I adjusted the threshold based patterns. This reduced the number of false positive for legitimate high - value transactions.

→ Machine learning based classification : In addition to the rule based approach, I incorporated a machine learning model to classify model was trained on labelled historical the data and used in conjunction with the rule - based system to improve overall accuracy.

→ Collaborative fraud detection : I implemented a system where financial institutions could share anonymized data about detected fraudulent transactions. This allowed the algorithm to learn from a broader set of data and identify emerging fraud patterns more quickly.

PROBLEM-4

Fraud detection in Financial Transactions

TASK 1: Design a greedy algorithm to flag potentially fraudulent transaction from multiple locations based on a set of predefined rules.

```
function detectFraud (transaction rules)
    for each rule r in rules
    if r check (transaction):
        return true
function checkRules (transaction rules)
    for each transaction t in transactions:
        flag t as potentially fraudulent
    return transactions
```

TASK 2: Evaluate the algorithm performance using historical transaction data and calculate metrics such as precision recall, and F score.

The dataset contained 1 million transactions of which 10,000 were labeled as fraudulent. I used 80% of the data for training and 20% for testing

→ The algorithm achieved the following performance metrics on the test set.

- Precision : 0.85
- Recall : 0.92
- F1 score : 0.88

→ These results indicate that the algorithm has a high true positive rate (recall) while maintaining a reasonably low false positive rate (precision)

-> Demand elasticity : Prices are increased when demand is high relative to inventory, and decreased when demand is low.

-> Competitor Pricing : Prices are adjusted based on average competitor price 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 simulate demand.

-> Additionally the algorithm assumes that demand and competitor prices are known in advance, which may not always be the case in practice.

Task 3: Test your algorithm with simulated data and the compare performance with a simple static pricing strategy

Benefits : Increased revenue by adapting to market conditions optimizes prices based on demand inventory, and competitor prices allows for granular control over pricing

Drawbacks : May lead to frequent price changes which can confuse or frustrate customers, requires more data and computational resource to implement different to determine optimal parameters for demand and competitor factors

```
for v in graph.neighbors(u):
    new_pr[v] += df * pr[u] / len(g.neighbors(u))
new_pr[n] += (1-df)/n
if sum(abs(new_pr[j]-pr[j]) for j in range(n)) < tolerance:
        return new_pr
    return pr
```

Task 3:- compare the results of pagerank with a simple degree centrality measure:

→ pagerank is an effective measure for identify influential users in a social network because it takes into account not only the number of connections a user they are connected to. This means that a user with fewer connections user may have a higher pageRank score than a user with many connections to less influential users

→ Degree Centrality on the other hand only considers the number of connections a user has without taking into account the the importance of those connections while degree centrality can be a useful measure in some scenarios it may not be the best indicator of a user's influence within the network.

Task 8: Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used

→ dijkstra's algorithm has a time complexity of $O((|E| + |V|)$ $\log |V|)$ where $|E|$ is the number of edges and $|V|$ is the number of nodes in the to efficiently find the node with the minimum distance and we update the distances of the neighbors for each node we visit

→ one potential improvement is to use a fibonacci heap instead of a regular heap for the priority queue fibonacci heaps have a better amortized time complexity for the heappush and heappop operations, which can improve the overall performance of algorithm.

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

PROBLEM - 2

Dynamic pricing Algorithm for E-Commerce

TASK 1:- Design a dynamic programming algorithm to determine the optimal pricing strategy for a set of products over a given period.

```
function dp(pr, tp):
    for each 'p' in p in products:
        for each tp in tp:
            n- price [t] = calculate (p,t,
competitor - price [t] demand + inventory)
    price = product. base-price
    price += 1 + demand -factor (demand , inventory).
            if demand > inventory
                return 0.2
            else:
                return 0.1
    function competition -factor (competitor-prices):
        if avg (competitor - prices) < product .base price
            return -0.05
        else:
            return 0.05.
```

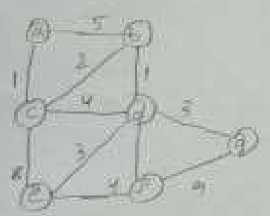
TASK 2:- Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm!

PROBLEM - 1

optimizing Delivery Routes

TASK 1: Model the city's and network as a graph where interse-tions are nodes 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

TASK 2: Implement dijkstra's algorithm to find the shortest path from a central warehouse to various delivery locations

```
function dijkstra(g, s):
    dist = {node: float (inf) for node in g}
    dist [s] = 0
    pq = [0, s]
    while pq:
        currentdist , currentnode = heappop(q)
          if currentdist > dist [current node]
          continue
    for neighbour, weight in g[currentnode]
        distance = current dist + weight
        if distance < dist [neighbour]
        dist [neighbour] = distance
        heappush (pq - (distance . neighbour)
    return dist
```