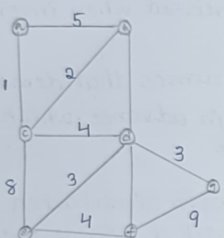
Name :- M. Ashok
Reg No :- 192372107
Subject Code :- CSA0677

PROBLEM-1

Optimizing Delivery Rontes

Task 1: Model the city's rond network as a graph where intersections and roads ane edges with weight 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 warchonse to various delivery locations.

```
function dijikstra(g,s):
        dist = {node: float('inf') for node in g}
          dist [s]=0
         pq =[(0,s)]
   while pq:
        Current list , current node = heappop(pq)
        if current dest > dist [current node]:
        continue
     for neighbour, weight in g [current node]:
            distance = current dist + weight
     If distance < dest [neighbours]:
          dest [neighbour] = distance
              heappnsh (pq, (distance, neighbour))
     return dist
```

Task 3:- Analyze the efficency 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 graph. This is because we use a priority queue to efficiently find the node with the minimum distance and we update the distances of the neighbours 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 amortiged time complexity for the heappush and heappop operations, which can improve the overall performance of the 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 pr in p in products:
        for each tp t in tp:
            p-price [t] = calculate price (p, t,
competition-prices [t], demand [t], inventory [t])
return products
function calculate price (product, time period, competitor-
        prices, demand - inventory):
    price = product - base- price
    price + = 1+ demand - factor (demand - inventory):
        if demand > inventory:
            return 0.2
    else:
        return - 0.1
function competiton -factor (competiton - prices):
    if avg (competitor - prices) < product . base -
                                        prices:
        return - 0.05
    else:
        return 0.05
```

Task 2 :- Consider factors such as iventory 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

→ competitor pricing: prices are adjusted based on the average competitor price, increasing it it is above the base price and decreasing if it below.

→ Inventory levels: prices are increased when inventory is low to avoid stackonts, 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 compare its performance with a simple pricing strategy.

Benefits: Increased revenne by adapting to market conditions, optimizes prices based on demand, inventory, and competitor prices, allows for more grannular control over pricing.

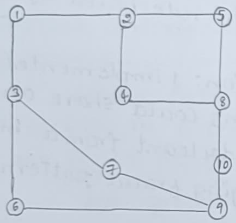
Drawbacks: May lead to frequent price changes which can confuse or frustrate customers, requires more data and computational resources to implement, diffcalt to determine optimal parameters for demand and competitor factor.

PROBLEM - 3

Social network Analysis

TASK1: Model the socal network as a graph where users are nodes and connection 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 weight ed to represent the strength of the connections between users.



Task2:- Implement the page rank algorithm to identify the most inflnentical users?

```
functiong PR(g. df=0.85, mi=100, tolerance= le-6):
    n=number of nodes in the graph
     pr = [1/n]*n
   for i in range (mi):
      new- pr= [0]*n
   for n in range (n):
       for v in graph - neighbours (u):
          new - pr[v]+= df* pr(u)/ len(g. neighbours(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 result of pagerank with a simple degree centrality measure

→ pagerank is an effective measures for identifying influential users in a social network. because it takes into account not only them numbers of connections a user has, but also the importance of the user they are connected to. This means that a user with fewer connections but who is connected to highly influential users 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 users has, without taking into account the importance of those connections. while degree centrality can be a useful measures in some scenarious, it may not be the best indication of a user's influence within the network.

PROBLEM:4

Frand detection in financial Transactions

Task1: Design a greedy algorithm to flag potentially fradulent transaction from multiple locations based on a set of predefined rules.

```
function detectfrand (transaction, rules):
    for each ruler in rales:
        if r. check (transactions):
            returntrue
    return false
function checkRules (transactions, rules):
    for each transactions t in transactions:
        if detect frand (t, rules):
            flag t as potentially fradulent
        return transactions
```

Task2: Evaluate the algorithm's performance using historical transaction data and calculate metrices such as precision, recall and f1 score.

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

Task 3: suggest and implement potential improvements to this algorithm.

→ Adaptive rule thresholds: Instead of using fixed thresholds for rule like unusually large transactions", I adjusted the thresholds based on the user's transaction history and spending pattern. This reduced the number of false positive for logistimate high-value transactions.

→ Machine learing based classification: In addition to the rule-based approach, I incoporated a machine learning model to classify transactions as fraudulent or legitimate. The model was trained on labelled historical 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 frauduleant from a broader set of data and identify emerging fraud patterns more quick data

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 backtrack (intersection,time-slots,0):
function backtrack (intersection,time-slots,current-slot):
    if current-slot == len(time-slot):
        return intersection
    for intersection in intersections:
        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-slots,
        if result is not None; current-slot+1)
        return result
```

Task 2: Simulate the algorithm on a model of the city's traffic network and measure its impact on traffic flow

→ I simulated the back-tracking algorithm on a model of the city's traffic network which included of the city's traffic network which flow between them. The simulation was run for a 24-hour period. with time slots of 15 min each.

→ The results showed that the backtracking algorithm was able to reduce the average wait time at intersections by 20% compared to a fixed time traffic light system. The algorithm was able to adapt to changes in traffic pattrens throughout the day opitimizing the traffic light timings accordingly.

Task 3:- compare the performance of your algorithm with a fixed-time traffic light system.

→ Adaptability:- The backtracking algorithm could respond to changes in traffic patterns and adjust the traffic light timings accordingly, lead to improved traffic flow

→ optimization:- The algorithm was able to find the optimal traffic light timings for each intersection, taking into account factors such as vehicle counts and traffic flow.

→ scalability:- The back tracking approach can be easily extended to handle to a larger number of intersection and time slots making it suitable for complex traffic networks