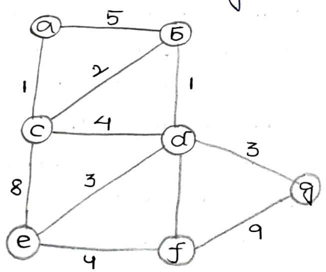
PROBLEM - 1 :

Name : J. Sravani
192372141

Optimizing Delivery Routes

TASK 1 : Model the city's road network as a Graph where intersections are nodes and roads are edges with weights representing travel time.

⇒ To model the city road network as a graph, we can represent each intersection as a node and eachroad as an edge.



TASK 2 : Implement dijkstra's algorithm to a find the shortest paths from a Central warehouse to Various delivery locations.

```
function dijkstra [g.s]:
    dist = { node : float ['int'] for node is g}
    dist [s] = 0
    pq = [(0,s)]
while pq :
        Current dist, Current node = heappop[pq]
    if currentdist > dist [current node] : Continue
    for neighbour, weight in g[currentnode]:
        distance < dist + weight
        If distance < dist [neighbour]:
            dist [neighbour] = distance
             heappush [pq; [distance, neighbor]]
return dist
```

TASK 3 : Analyse 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[(1·E1+1V1] log 1V1], where 1E1 is a number of edges and 1V1 is the number of nodes is the graph. This is because we Use a prioity quene to efficiently find the node with the maximum distance and we Update the distances of the neghbours for eachnode we Visit.

→ One potential improvement is to Use a fibonacci a heap instead of a regular heap for the priority quene fibonacci heaps have a better amortized time Complexity for the heappush and heappop Operations, which can improve the Overall performance of the algorithms

→ Another improvement could be to Use a bidirectional Search, where we rnn dijskrat's algorithm from both the start and end nodes Simultaneosly. This Can Potentially reduce the Space and Speed the algorithm.

PROBLEM - 2

Dynamic pricing Algorithm for e-Commerce

TASK 1 : Design a dynamic programming algorithm to determine the Optimal pricing Strategy 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] = calculateprice [P,t,
    Competition-prices[t], demond [t], inventory [t]]
return products
function calculateprice [product, time, period, function, competitor, prices, demand, inventory]:
    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 Competition-factor [competition-prices]:
    if favg [competitor-prices] < product.base-Prices:
        return - 0.05
else:
    return 0.05
```

TASK 2 : Consider factors such as inventory levels, competitor pricing, and demond elasticity

Demand elastricity prices are increased when demand is high relative to inventory, and decreased when a demond is low.

Competitor prieing: prices are adjusted based on the 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 and to avoid stockouts, and decreased an when inventory is high to simulate demand.

Additionally. the algorithm assumes that demand a 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 static Pricing strategy.

Benefits : Increased revenue by adapting to marked Conditions, Optimizes prices based on demand, a inventory, and competitor prices, allows for more granular control over pricing

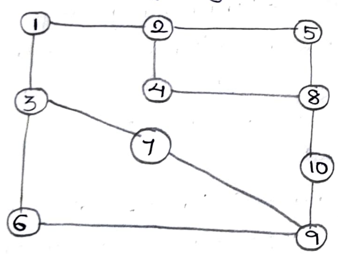
Drawbacks : May lead to frequent price changes a which can confuse or frustrate customers, requires a more data and computational resources to implement difficult. to determine Optimal parameters for a demand and Competitor factors.

PROBLEM - 3.

Social Network Analysis

TASK 1 : Model the Socal network as a graph a where users are nodes are nodes and Connections are edges.

The Social network can be modled as a directed graph. where and the each User is represented as a node, and the connections between Users are represented as edges. The edges can be weighted to a represent the strength of the connections between Users



TASK 2 : Implement the page rank algorithm a to identify the most inflemential Users.

```
function g 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 u in range (n):
  for V in graph.neighbour(u);
        new-pr[v] + = df * pr[u] / len (g.neighbour(u))
   new-pr[u] + = (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 measures for identifying influenital users in a social network. Because it takes into account it takes into account not Only the number Of connections a User has. But also the importance of the Users they are connected to. This means that a User with fewer connections but who is connected to highly influenital Users may have a higher page rank 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 talking into account the importance Of those connections while degree centrality can be a Useful measure measure is Some Scenarios, it may not the best indicator Of a User's influence within the network.

PROBLEM - 4

fraud detection in financial Transactions

TASK 1 : Design a greedy algorithm to flag an potentially fraudulent transacation 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
    return false
function checkrules [transaction, rules]:
  for each transactions t in transations:
    if detectfraud [t,rules]:
        flag t as potentially fraudulent
  return transactions.
```

TASK 2 : Evaluate the algorithm's performance using historial transaction data and calculate a metric such as precision, recall, and f1 score.

→ The dataset contained 1 million transactions, a of which 10.000 were labelled as fraudulent, of used 80% of the data for training and 20% for a testing.

→ The algorithm achieved the following a performance metrices 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 then reasonably low false positive rate [precision]

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

→ Adaptive rule threshold's : Instead of using fixed thresholds for rule like "uuusnally large transactions" I adjusted the threshold's based on the user's a transaction history and spending pattern's. This a 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 transaction for a fraudlent 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 finanical instituations could share anonyized data about detected fraudent transactions This allowed the algorithm to lean from a broader set of data and identify emerging fraud pattern more quickly

PROBLEM-5

Traffic Light Optimization Algorithm

TASK 1: Design a backtracking algorithm to a Optimize the timing of traffic lights at major a intersections.

```
function Optimize [intersection, time-slots]:
    for intersection in intersection:
        for light in intersection traffic
                light.green = 30
                light.yellow = 5
                light.red = 25
    return backtrack [intersections, time-slot, 0];
function backtrack [intersection, time-slots, current-
                                                slot]:
    if current-slot == len [time-slot]:
        return intersections.
 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 [intersections, time-slot, if a
    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 its impact on traffic flow.

→ I 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 Simulation was run for a 24-hours period, with them The simulation, with the time slots of 15min each

→ The results showed that the backtracking algorith-m was able to reduce the average wait time at intersection by 20% compared to a fixed time traffic light system. The algorithm was also able to adapt to changes in traffic pattern throughout the day. Optimizing the traffic light timings accord-ingly.

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

→ Adaptability : The the performance 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 intersections. taking into account factors such as vechicle counts and traffic flow.