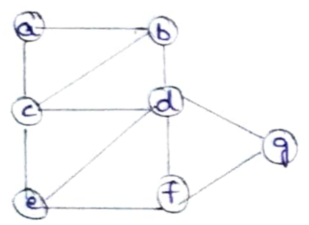
Name : P. Sisindri
Reg No : 192324122

# Problem - 1

## Optimising 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's road network as a graph we can represent each intersection as a node and each node as an edge




The weights of the edges can represent the travel time b/w intersections

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

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

Task 3: 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 no. of edges and $|V|$ is the no. 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

→ 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

→ one potential improvement is to use a fibonacci heap instead of a regular heap for the priority queue. Fibonacci heaps have better amortized time complexity.

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
            foreach 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 - 0.05
       else
            return 0.0;
```

Task 2: 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

-) competitor pricing prices are adjusted based on the average competitor price, increasing 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

Task 3 Test your algorithm with stimulated data and compare its performance with a simple static pricing strategy

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

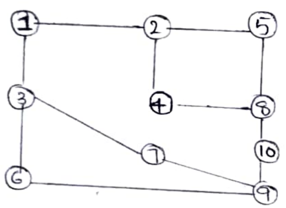
Drawbacks: May lead to frequent (customers) price changes which can confuse or frustrate customers, requires more data and computational resources to implement, difficult to determine optimal parameters for demand for competitor factors

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 b/w users.




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

```
functioning PR(g, df = 0.85, mi = 100, tolerance = 1e-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.neighbour(u))
            new-pr(u) += (1-df) / n
        if sum (abs (new - pr[i] - pr[i]) for j in range
        (n) < tolerance :
            return new-pr
    return pr.
```

**Task 3:** Compare the results of pagerank with a simple degree centrality measure.

-> Page rank is an effective measures for identifying influential users in a social network because it takes into account not only the no. 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 influential users may have a higher PageRank score than a user with many connections to less.

-> Degree centrality on the other hand, only considers the no. of connections a user has without taking in to account of the importance of those connections while degree centrality can be a useful measure in some scenarios, it may not be the best indicator of user's influence within the network

Problem - 4:

Fraud detection in financial transactions

Task 1: Design a greedy algorithm to flag potentially fraudulent transactions from multiple locations, based on a set of predefined rules

```
function detectfraud (transaction, rules):
    for each rule r in rules:
        if r.check (transactions):
            return true
    return false.

function check rules (transaction, rules):
    for each transaction in transactions:
        if detectfraud (t, rules):
            flag t as potentially fraudulent
    return transactions.
```

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

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

→ The algorithm achieved the following metrics on the test set

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

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 transactions history and spending patterns. This reduced the no. 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 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 fraudulent transactions. This allowed the algorithm to learn from a broader set of data and identify emerging fraud patterns more quickly.

Problem - 5

## Traffic light optimisation algorithm.

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

```
function optimise (intersections, time-slots):
    for intersection in intersections:
        for light in intersection. traffic
            light. greeen = 30
            light. yellow = 5
            light. red = 25
    return backtrack (intersections, time-slot, 0):
function backtrack (intersections, time-slots, current_
                                            slot ):
    if current_slot = = len(time-slots):
        return intersections
    for intersections in intersections:
        for light in intersections. 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 )
    return result.
```

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

→ I simulated the backtracking algorithm on a model of the city's traffic network, which included the major intersections and the traffic flow b/w them. The simultaneously 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 also able to adapt to changes in traffic patterns

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.

Optimisation: The algorithm was able to find the optimal traffic light timings for each intersection taking into account factors such as vehicle counts

Scalability: The backtracking approach can be easily extended to handle a larger no. of intersection and time slots, making it suitable for complex traffic networks.