Assignment - 01

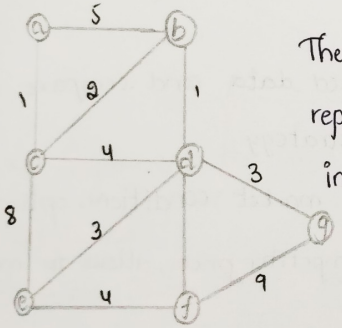
P. Sumalavasavi
RegNo: 192372149

Problem - 1

Optimizing Delivary 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 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 delivary locations.

```
function dijkstra(g,s):
    dist = {node: float('inf') for node in g}
    dist[s]=0
    pq = [(0,s)]
    while pq:
        currentdist,currentnode = heappop(pq)
        if currentdist > 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 improvement 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 miniumum distance and we update the distances of the neighbour 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 time complexity, which can improve overall performance.

→ Algorithm improvement could be to use a bidirectional search, where we run dijkstra's algorithm from the both the strat node and end node simultaneously. This can potentially reduce the search space and speedup 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 competition_factor (compititor_prices):
    if avg (competition_product) < product.base_prices:
        return -0.05
    else:
        return 0.05
```

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 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 [low to avoid] x high to simulate.

→ 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 static pricing strategy.

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

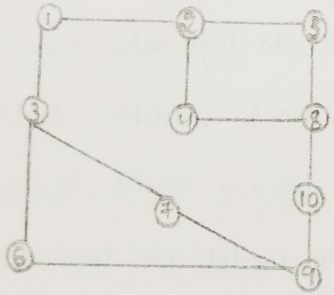
Drawbacks: May lead to frequent price changes which can confuse or frustrate customers, requires more data and computational resource to implement, difficult to determine optimal parameters for demand and competitor factors.

PROBLEM-03

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 modelled 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 connection between users.



TASK-2: Implement the page rank algorithm to identify the most influtential user.

```
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 u in range (n):
          for v in graph.neighbours(u):
          new-pr[v]+= df * pr[u]/ len(g.neighbors(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 page rank with a simple degree centrality measure.

→ Page Rank is an effective measures for identity influential user in a social network because it takes into account not only the number of connections a user has, bud also the importance of the users that they are connected to. This means that a user with fewer connections but who is connected to highly influential users.

→ Degree centrality, on the other hand, only considers the number of connections a user has, without taking into acount the importance of those connections. While degree centrality can be useful measure in some scenarious, it may not be the best indicator of a users influence within the network.

PROBLEM 04

Fraud detection in financial Transactions

TASK: 1: Design a greedy algorithm to flag potentially fraudlent transaction 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 checkRules (transactions, rules):
    for each transaction t in transactions:
        if detectfraud (t, rules):
            flag t as potentially
                fradulent
    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 fraudent. 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 algorithm has a high true positive rate [recall] while maintaing a reasonbly low false positive rate which is also called is precision

TASK-3: Suggest and implement potential improvement to this algorithm.

→ Adaptive rule thresholds: Instead of using fixed thresholds for rule like "unsually large transactions" I adjusted the thresholds based on the users transaction history and spending patterns. This reduced the number of false positive for legitimate high-value transtions.

→ Machine learning based classification: In addition to the rule-based approach, I incorpoted 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 fraudelent transactions. This allowed the algorithm to learn from a boarder set of data and identify emerging fraud patterns more quickly.

PROBLEM : 05

Traffic light optimization algorithm:

TASK 1: Design a backtracting algorithm to optimize the timing of traffic lights at major intersections:

```
function optimize (intersections, time_slots):
    for intersections in intersections:
        for light in intersection.traffic
            light.green =30
            light.yellow=5
            light.red =25
    return backtrack
function backtrack (insertions, time_slots, current-slot):
    if current_slot == len(time_slots):
        return intersections
    for intersection in intersections:
        for light in intersection.traffic:
            for green in [20, 30, 40]:
                for yellow in [3,5,7]:
                    light.green = green
                    light.yellow= yellow
                    light.red=red
    result= backtract( intersection, time_slots, current_slot+1)
    if result is not None:
        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 the major intersections and the traffic flow between them. The simulation was run for 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 able to reduce the average wait time at intersections by 20%.

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

→ Adaptibility :- 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.

→ Scalability: The backtracking approach can be easily extended to handle a larger number of intersection and time slots, making it suitable for complex traffic networks.