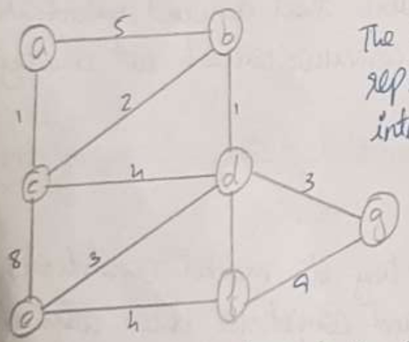
M.M. Naveen
192311166
CSA0766-DAA
Assignment-1

Problem: 1

Optimizing Delivery Routes

Task 1: Model the city's road network as a graph where intersection are nodes and roads are edges with weight representing travel time:

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



The weight of the edge can represent the travel time between intersection.

Task 2: Implement dijikstra's algorithm to find the shortest path from a central warehouse to various delivery location

```
function dijkstra (g,s):
    dis = { node : float ('int') for node in g }
    dist[s] = 0
    Pq = [(0,s)]
    while Pq:
        (currentdist, currentnode, heappop(Pq)
        if currentdist > dist[currentnode]:
            continue
        for neighbour weight in g[currentnode]:
            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 algorithm 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 nodes with the minimum distance and we update the distance 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 heap have a better amortized time complexity for the heappush and heappop operation, which can improve 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 Product over a given Period.

```
function dp(Pr, tp):
    for each Pr in P in Product:
        for each tp t in tp:
            P.Price[t] = calculate Price(P,t)
competition-Prices(t). demand [t], inventory[t]
return Product
function ~~Product~~ calculate (Product, inventory):
        Price t= 1+demand. factor(demand, inventory):
        Price = Product.base.Price
        if demand > inventory :
            return 0.2
        else :
            return 0.1
function competition factor ( competitor_Price ):
    if avg(competitor_Prices) < Produce.base_Prices:
        return-0.05
    else :
        return 0.05
```

Task 2: Consider factor such as inventory level, competitor Pricing and demand elasticity in your algorithm

-> Demand elasticity: Price 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: Price are increased when inventory is low to avoid stockouts and decreased when inventory is high to Simulate demand

-> Additionally, the algorithm assume that demand, ~~which~~ and competitor Price are Known in advance, which 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 condition, optimize Prices based on demand inventory and competitor Prices, allow for more granular control over Pricing

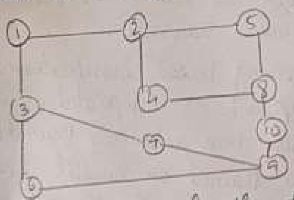
Drawback: may lead to frequent Price changes which can confuse (or) frustrate Customer, require more data and computational resource to implement, difficult to determine optimal Parameter for element and competitor factor.

Problem -3

Social network Analysis

Task 1: Model the social network as a graph where user 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 connection between user are represented as edges. The edges can be weight to represent the strength of the connection between user



Task 2 Implement the Page rank algorithm to identify the most influential user.

```
function 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.neighbour(u):
            new-Pr[V]= df * Pr[n]/len(g.neighbour(n))
        new -Pr[n] +=(1-df)/n
        if sum (abs(new-Pr[i] -Pr[i] for i in range
          (n) < tolerance
```

Task 3 Compare the result of Pagerank with a simple degree centrality measure.

→ Page Rank is an effective measure for identifying influential user in a social network because it takes into account not only the number of conection a user has but also the importance of the user are connected to ~~with fewer connection~~ This means that a user with fewer connection but who is connected to highly influential user may have a higher Page Rank score than a user with many connection to less influential user.

→ Degree centrality on the other hand only consider the number of connection a user has without taking into account the importance of those connection. 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.

Problem 4

Fraud detection in financial Transaction.

Task 1: Design a greedy algorithm to flag Potentially fraudulent transaction from multiple location 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 transaction t in transaction:
        if detectfraud(t, rules):
            flag t as Potentially fraudulent
    return transaction.
```

Task 2: Evalute the algorithm Performance using historical transaction data and calculate metrics such as Precision, recall and score.

The ~~dataset~~ dataset contained 1 million transaction 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 Perforance metrices on the test set

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

-> These result indicate that the algorithm has a high true Positive rate [recall] while maintaining a resonably low false Positive rate [Precision]

Task 3: Suggest and implement Potential improvement to this algorithm.

-> Adaptive rule thresholds: Instead of using fixed thresholds for rule like "unusally large transaction", I adjusted the thresholds based on the user transtaction history and spending Pattern. This reduced the number of false Positive for legitimate high value transaction

-> Machine learning based classification: In addition to the rule based approach, I incorporated a machine learning model to classify transaction 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 insititutions could share anonymized data about detected fraudulent transaction. This allowed the algorithm to learn 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 optimize the timing of traffic light at major intersections

```
function optmize (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_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-slots, current_slots)
    if result is not None:
        return result.
```

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

-> 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. The Simulation was run for a 24-hour Period, with time Slots of 15min each

-> The result showed that the backtracking algorithm 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 toadapt to changes in traffic Pattern throughout the day. optimizing the traffic light timing accordingly.

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

-> Adaptability: The backtracking algorithm could respond to change in traffic Patterns and adjust the traffic light timing accordingly. lead to improved traffic flow.

-> Optimization: The algorithm was able to find the optimal traffic light timing for each intersection, talking into account factor such as vehicle counts and traffic flow

-> Scalability: The backtraking approach can be easily extended to handle a large number of intersection and time slots, making it suitable for complex traffic network.