ASSIGNMENT - 1

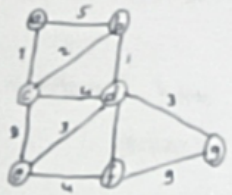
G. Ayyappa venkata sai
192372111

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 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 edge.



the weights of the edge's can represents the travels time between intersections.

Task 2: Implement dijkstra's algorithm to find the shotest paths from a central ware house to various delivery locations

```
function dijkstra(g,s)
    dist = [node : float("inf") for node in g}
    dist[s] = 0
    pq : [(0,s)]
    while pq:
        currentdist, current node = heap pop(pq) if
currentdist > dist[current node]:
        continue
    for neighbour weight in g[current node]:
        distance = currentdist weight
        If distance < dist [neighbour]
            dist [neighbour] = distance
            heap Push (pq. (distance. neighbour))
    return dist
```

Task 3: Analyse the efficiency of your algorithm and discuss any potential improvements or allernative 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 minimum distance and we update the distances of the neighbors for each node we visit

-> one potential improvement is to use a fibonaci heap instead of a regular heap for the Priority queue. Fibonna heap have a better amortized time complexity for the heappush and heappop operations, which can prove the overall Performance of the algorithm.

-> Another improvement could be to use a bidirectional search, where we run dijkstra's bidirectional search 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(PT, tP):
    for each PT in P in products:
        for each tp in tP:
            P.Price[t] = demand[t], inventory[t]
    return products
function calculatePrice(product, timeperiod, 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 competitor_factor(competitor_prices):
    if avg(competitor_prices) < 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: Price are adjusted based on the average competitor Price, increasing if 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 demand

-> Additionally, the algorithm assumes that demand and competitor Prices are know 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 resources to implement, difficult to determine optimal parameters for demand and 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 b/w users are representation as edges. The edge can be weight to represent the strength of the connections b/w users

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

```
function 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.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.

→ Page Rank in an effective measures for identying influential users in a social network because it takes into account not only the number of connections a user they are connected to this mean that a user with fewer connections but who is connected to higher PageRank score than a user with many connections to less influential.

→ Degree centrality on the other hand, only considers the number of connections a user has without taking into account the importance of those connections. 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 transactions

TASK 1: Design a greedy algorithm to flag potentially fraudulent 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 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 labeled as fraudulent & 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
- F1score: 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 "large transaction", I adjusted the threshold based on the user's transaction history and spending patterns. This 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 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: 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 optimization Algorithm

TASK 1 :- Design a backtracking algorithm to otimize the timing of traffic lights at majon intersections

```
function optimize (intersections, time_slots):
    for intersection in intersections:
        for light in intersection.traffic
            light.green = 30
            light.yellow = 5
            light.red = 25
    return backtrack (intersections, time_slots, 0):

function backtrack (intersections, 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]:
                    for red in [20,25,30]:
                        light.green = green
                        light.yellow = yellow
                        light.red = red
    result = backtrack (intersections, time_slot, 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 btw them The simulation was run for an 24-hour period, with time slots of 15 min each

-) The results should that the backtracking algorithm was able to reduce the awerage wait time at intersections by 20% compared to a fited time traffic light system The algorithm was also able to adapt to changes in traffic pattens the day. optimizing the traffic light timings accordingly.

TASK 3 :- compare the performance of your algorithm with a fited 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 inter section, taking into account factors such as vehicle count and traffic flow

-) scalability :- The backtracking approach can be easily extended to handle a langer number of intersection and time slots, making it suitable for complex traffic networks.