V.VishwaTej
192321124

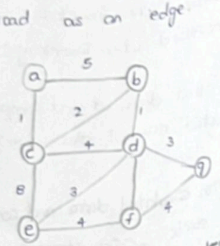
Problem-1

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'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 paths from a central warehouse 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, currentnode = heap pop(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 improvements 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 node with the minimum distance and we update the distances of the neighbours 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 amortized time complexity for the heappush and heappop operations, which 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 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), demand[t] invetory [t])
return products.
function calculateprice (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 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: 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 high to simulate demand.

→ 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.

Benfits: Increased revenue by adapting to market Conditions, Optimizes prices based on demand, inventory and competitor prices, allows for more granular control over pricing.

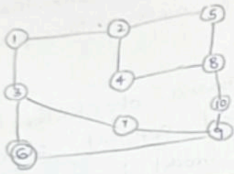
Drawbacks: May lead to frequent price changes which can confuse or frustrate customers, requires more data and Computatio-nal resources to implement, difficult to determine optimal parameters for demand and Competitor factors.

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 Connection between users are represented as edges can be weighted to represent the strength of the Connections between 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.neighbours(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 is an effective measures for identifying influential users in a social network because it takes into account not only the number of connections a user has but also the importance of the user with fewer Connections but who is connected to highly influential 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 numbers of connections as 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 predefiened rules

```
function detectfraud (transactions, rules):
    for each rule r in rules:
        if r.check (transactions):
            return true
    return false
function check Rules (transactions, rules):
    for each transaction t in transactions:
        if detect fraud (t rules):
            flag t as potentially fraudulent
    return transactions.
```

Task 2: Evaluate the algorithm's performance using historical transcation data and calculate metrics such as precision, recall, and f_1 score.

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

→ The algorithm achieved the following performance metrices on the test set.

- precision: 0.85
- Recall: 0.92
- F_1 score: 0.88

→ These results indicate that the algorithm has a high true positive rate [recall] while maintaining a reasonably low false positives rate [precision]

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

→ Adaptive rule threesholds: Instead of using fixed threeshold for rule like unusually large transactions", I adjusted the threesholds based on the user's transactions 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 in incorporated a machine learning model to classify transactions as fradntd or legitimate. The model was trained on labelled historical data.

→ Collaborative fraud detection: Implemented a System where financial institutions could share anonymized data and identify.

Problem - 5

Traffic light Optimization Algorithm.

Task 1: Design a backtracking algorithm to optimize the timing of traffic lights at major 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.
    if current_slot == len (time_slot):
        return intersections
    for intersections 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_slots,
    if 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 on traffic flow

→ I simulated the back tracking algorithm on a model of the city's traffic network the traffic flow between them. The simulation 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 day optimizing the traffic light timings accordingly.

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 lights 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 vechile counts and traffic flow.

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