

ASSIGNMENT

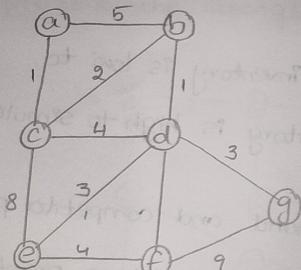
V. Gianga Deepika
192325072

PROBLEM-1:

Optimizing Delivery Routes

Task-1: Model the city's road network as a graph where intersections are nodes and roads where 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, 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
```

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 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 minimum distance, and we update the distance 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.

Output formats:

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], inventory[t])

return products

function calculateprice(product, timeperiod, competitor_prices, demand, inventory):

price = product.base_price + demand_factor(demand, inventory) * (product.base_price - competitor_prices[timeperiod])

price += demand_factor(demand, inventory) * (product.base_price - competitor_prices[timeperiod])

if demand > inventory:

return 0.2 * price

else:

return -0.1 * price

function competition_factor(competition_prices):

if avg(competition_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 decreased 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.

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

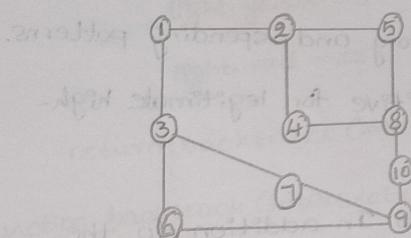
Drawbacks: May lead to frequent price changes which can confuse or frustrate customers, requires more data and computational resource to implement, difficult to determine optimal parameters for demand and competitor factors.

PROBLEM-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 between users.



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

functioning PR(g, df=0.85, m=100, tolerance=1e-6):

n = number of nodes in the graph

$$pr = [1/n] * n$$

for i in range(m):

$$\text{new-pr} = [0] * n$$

for u in range(n):

 for v in graph.neighbors(u):

$$\text{new-pr}[v] = df * \text{pr}[u] / \text{len}(g.\text{neighbours}(u))$$

$$\text{new-pr}[u] += (1-df) / n$$

if sum(abs(new-pr[i] - pr[i])) for i 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 measure 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 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 page rank score than a user with many connections to less influential users.

→ 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 Transaction:

Task-1: Design a greedy algorithm to flag potential 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(Transaction,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 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 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 unusually large transactions. I adjusted the thresholds 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 classifications: 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.

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 (intersection, time-slots):

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 [20,25,30]:

 light.green = green

 light.yellow = yellow

 light.red = red

result = backtrack (intersections, time-slots, current-slot+1)
if result is not None:
 result 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 intersection and the traffic in flow between them the simulations 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 intersection by 20% compared to a fixed.

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.

→ **Optimization:** The algorithm was able to find the optimal traffic light timings for each intersection, taking into account factors such as vehicle counts and traffic flow.