DAA-Assignment

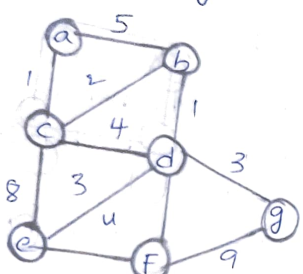
Name:- S.Susmitha
Course code:- CSA0766
192372140

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 a node 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 froms 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 [current node]:
       continue
for neighbour, weight in g[currentnode]:
      distance = currentdist + weight
     If distance < dist [neighbour]:
       dist [neighbour] = distance
       heappush (pq.(distance,neighbor))
   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 efficiency find the node with the minimum distance of the neighbors for each node we visit.

* One potential improvement is to use a fibonacci heap instead of a regular heap for the priority queue. Fibonnacci 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.Prices[t] = calculateprice (p, t,
  competitor-prices[t], demand[t], inventory[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 : Price are increased when inventory is Low to avoid stockonts and decreased when inventory is high to simulate demand.
* Additionally, the algorithm assumes that demand and competitory 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 pric allows for more granular control over pricing.

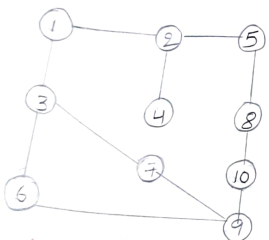
Drawbacks : May lead to frequent price changes which car confuse or frustrate customers, requires more data and comp ational resources to implement, different to determine optim Parameters for demand and competitor factors.

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:

```
function g 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.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 Pagerank with a simple degree centrality measure.

* PageRank is an effective measures for identying 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 but who is connections users may have a higher pageRank score than a user with many connections to less influenential 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

Frand detection in financial Transactions

Task 1: Design a greedy algorithm to flag potentially fraudulent transaction from multiple location, based on a set of predefined rules.

```
function detectfrand(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 detectfrand(t, rules):
            flag t as potentially frandulent
    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 0,000 were labeled as frandulent. 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 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 "usasually 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 classification: In addition to the rule-based approach, I incorporated a machine learning model to classify transactions as frandulent or legitimate. The model was trained on labelled historical data and used in conjunction with the rule-based system to imppove overall occurancy.

Collaborative frand detection: I implemented a system where financial institutions could share anonymized data about detected frandulent transactions. This allowe the algorithm to learn from a broader set of data and identify emerging frand patterns more quickly

Problem-5

Traffic light optimization Algorithm

Task 1 : Design a backtracking 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-slot):
    if current_slot = len(time-slots):
        return intersections.
    for intersection 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-slot+1)
        if result is not none:
            return result
        return none.
```

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 a 24-hour period, with time slots of 15 min each.

* The result 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 algorithm was also able to adapt to change in traffic light timings accordingly.

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

* Adaptability : The back-tracking 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 vehicle counts and traffic flow.
* 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.