def count_pairs(nums,k):

Assignment

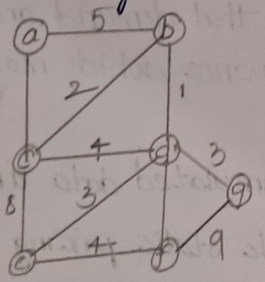
Name : A.Sravanthi
Reg no : 19032507

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 represen the travel Each intersection as a node and each node as an edge.



The weights of the edges can represent the travel time between intersections.

TASK 2 : Implement dijkstra's algorithm to find the shotest paths from a central warehouse to various delivery locations

```
function dijkstra(g,s);
    dist = {node:float('inf') for node in g}
    dist[s] = 0
    P,Q = [(0,s)]
    while PQ:
        currentdist, currentnode = heappop(PQ)
        if currentdist > dist[current node]:
            for neighbour, weight in g[current node]:
                distance = currentdist + weight
                if distance < dist[neighbor]
                    dist[neighbour] = distance
                    heappush(pq, (distance, neighbor))
    return dist
```

TAST 3 : Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithm that could be used

* Dijkstras algorithm has a time complexity of O((|E|+|V|) is the number of edges and |V| is the number of nodes in the graph. This is because we use a perority queue to efficiency find the node with the minimum distance, and we update the distance.
* 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 prin 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, time period, competitor-
                        prices, demand, inventory):
price = product.base price
price += if demand_factor(demand, inventory):
    if demand > inventory:
        return -0.2
    else:
        return -0.1
function competition_factor(competition_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

* 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

def

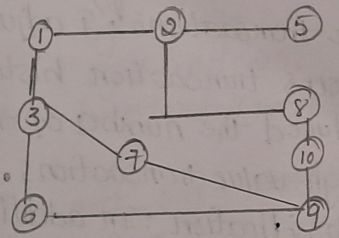
Name : A. Sravanthi

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



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

```
-functiong PR(g, df=0.85, mi=100, tolerance=0.0]):
    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[w] / len(g.neighbors(u)):
    new-pr[n] += (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.

* pagerank is an effective measures for identifying influential users they 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 pagerank 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 best indicators of a users influence within the network.

PROBLEM 4

Frand detection in financial transaction

TASK 1 : Design a greedy algorithm to flag potentially fradulent transaction of from multiple locations, based on a set of predefined rules.

```
function defectfrand (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-frand (t, rules):
            flag t as potentially fradulent
return transactions
```

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

* functioning the dataset contained 1 million transactions, of which 10.000 were labeled as fradulent of used 80% of the data for training and 20% for testing
* The algorithm achieved the following performance metric on the test tset.
  * precision : 0.85
  * Recall : 0.92
  * Fscore : 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 threshold : Instead of using fixed thresholds for rule like usually large transactions, I adjusted the thresholds based on the user's transaction history and spending patterns. The reduced the number of false positive for legistimate high-value transactions.
* Machine learning based classification : In addition to the rule-based approach, I incorporation rated a machine learning model to classify transactions as fraudulent or legistimate. The model was trained on labelled historical data and used in conjuntion with the rule-based system to improve overall accuracy.
* Collaborative froud detection : I implemented a system where financial institutions could share anonymized data about detected fraudulent transactions. This allowed the algorith to learn from a broader set of data and identify emerging fraud more quickly

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, current_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 (intersection, 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 citys traffic network, which included the major intersections and the traffic flow between them. The simulation was run for 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 time traffic light system. The algorithm was also able to adapt to changes in traffic patterns throughout 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 light timings accordingly, lead to improved traffic flow
* Optimization: The algorithm was able to find the optimal traffic light timings accordingly for each intersection taking into account factors such as vehicles count and traffic flow.