**CSA 0674-ASSIGNMENT:-**

Problem 1: Optimizing Delivery Routes (Case Study)

**Scenario:** You are working for a logistics company that wants to optimize its delivery routes to minimize fuel consumption and delivery time. The company operates in a city with a complex road network.

**Tasks:**

1. Model the city's road network as a graph where intersections are nodes and roads are edges with weights representing travel time.

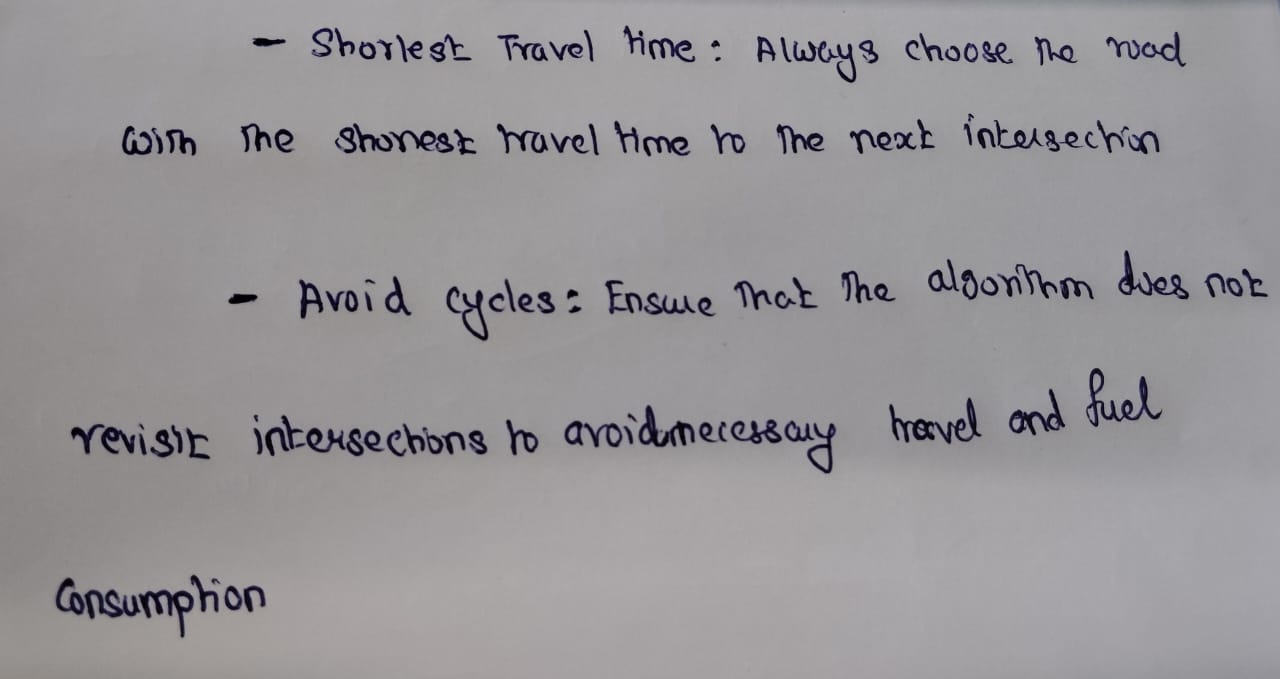
2. Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations.

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

**SOLUTION:-**

**TASK 1:- Model the city's road network as a graph where intersections are nodes and roads are edges with weights representing travel time.**

In this Problem**,**We aim to optimize delivery routes to minimize fuel consumption and delivery time in a city with a complex road network. We will use a greedy algorithm to select the next best step at each intersection based on predefined rules**.** A greedy algorithm makes a locally optimal choice at each step with the hope of finding a global optimum. In this context, the algorithm will choose the next intersection to visit based on the shortest travel time from the current location.



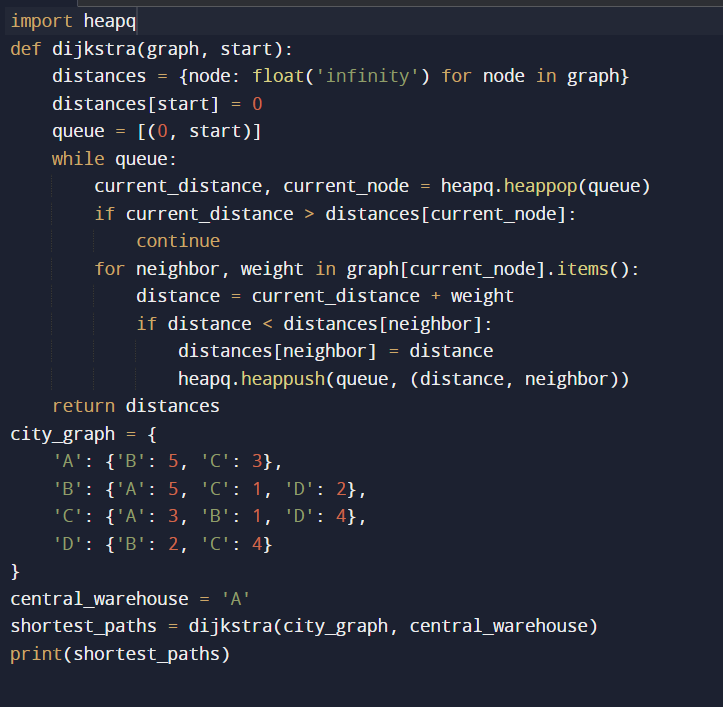
**TASK 2:- Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations..**

Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations. The city's road network is modeled as a graph where intersections are nodes and roads are edges with weights representing travel time. The algorithm initializes distances from the warehouse to all delivery locations, setting the initial distance to the warehouse itself as zero and all other distances as infinity. Using a priority queue, the algorithm explores nodes by selecting the one with the shortest known distance at each step. It updates the distances to neighboring nodes if a shorter path is found and continues this process until all nodes are processed. The program then retrieves and displays the shortest paths and their respective travel times from the warehouse to each delivery location.

**Steps to Implement Dijkstra’s Algorithm**

1. **Initialize Distances:** Set the initial distance to the source node (warehouse) as 0 and all other nodes as infinity.
2. **Initialize Priority Queue:** Use a priority queue (min-heap) to keep track of nodes to explore based on their current shortest distance.
3. **Update Distances:** For each node, update the distances to its neighboring nodes if a shorter path is found.
4. **Repeat:** Continue the process until all nodes are processed.
5. **Retrieve Paths:** Store the shortest paths and distances from the source node to all other nodes.

**IMPLEMENTATION:-**



**OUTPUT:-**



**PSEUDOCODE:-**

Algorithm Dijkstra(graph, start):

Input: graph - a dictionary where keys are nodes and values are lists of tuples (neighbor, weight)

start - the starting node (central warehouse)

Output: distances - a dictionary with the shortest distance from the start to each node

shortest\_path\_tree - a dictionary representing the shortest path tree

// Initialize distances and priority queue

distances = {node: infinity for each node in graph}

distances[start] = 0

priority\_queue = [(0, start)] // (distance, node)

shortest\_path\_tree = {}

// Process nodes in priority queue

while priority\_queue is not empty:

current\_distance, current\_node = extract\_min(priority\_queue)

// Skip if the current distance is longer than the known shortest distance

if current\_distance > distances[current\_node]:

continue

// Update distances to neighbors

for each neighbor, weight in graph[current\_node]:

distance = current\_distance + weight

// If a shorter path is found, update the distance and priority queue

if distance < distances[neighbor]:

distances[neighbor] = distance

insert(priority\_queue, (distance, neighbor))

shortest\_path\_tree[neighbor] = current\_node

return distances, shortest\_path\_tree

// Function to extract the minimum element from the priority queue

Function extract\_min(priority\_queue):

// Extract and return the element with the minimum distance

return heapq.heappop(priority\_queue)

// Function to insert an element into the priority queue

Function insert(priority\_queue, element):

// Insert the element into the priority queue

heapq.heappush(priority\_queue, element)

// Example usage

graph = {

'A': [('B', 5), ('C', 10)],

'B': [('A', 5), ('C', 3), ('D', 8)],

'C': [('A', 10), ('B', 3), ('D', 2), ('E', 7)],

'D': [('B', 8), ('C', 2), ('E', 4)],

'E': [('C', 7), ('D', 4)]

}

start\_point = 'A'

distances, shortest\_path\_tree = Dijkstra(graph, start\_point)

// Output the results

print("Shortest distances from the central warehouse (A):")

for each node, distance in distances:

print("Distance to ", node, ": ", distance)

print("\nShortest paths from the central warehouse (A):")

for each node in shortest\_path\_tree:

path = []

while node is not null:

path.append(node)

node = shortest\_path\_tree.get(node, null)

print("Path to ", path[-1], ": ", join(" -> ", reverse(path)))

**TASK 3:- Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.**

**Efficiency Analysis of Dijkstra's Algorithm**

1. **Time Complexity:**
   * **Initialization:** Setting up the distances dictionary and the priority queue takes O(V)O(V)O(V), where VVV is the number of vertices (nodes).
   * **Priority Queue Operations:** Each vertex is inserted into the priority queue once, and each insertion and extraction operation in a min-heap takes O(log⁡V)O(\log V)O(logV). This results in O(Vlog⁡V)O(V \log V)O(VlogV) for all insertions and extractions.
   * **Edge Relaxation:** Each edge is examined and relaxed at most once, resulting in O(E)O(E)O(E) edge relaxations, where EEE is the number of edges. Each relaxation involves a priority queue operation, so this takes O(Elog⁡V)O(E \log V)O(ElogV) in total.

Combining these, the overall time complexity of Dijkstra's algorithm is O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV). For dense graphs, where EEE can be close to V2V^2V2, the time complexity approximates

O(V2log⁡V)O(V^2 \log V)O(V2logV). For sparse graphs, where EEE is much smaller than V2V^2V2, it approximates O(Vlog⁡V)O(V \log V)O(VlogV).

1. **Space Complexity:**
   * **Graph Representation:** Using an adjacency list, the space complexity for storing the graph is O(V+E)O(V + E)O(V+E).
   * **Auxiliary Storage:** The distances dictionary, priority\_queue, and shortest\_path\_tree each require O(V)O(V)O(V) space.

**Potential Improvements:-**

1. **Using Fibonacci Heaps:**
   * A Fibonacci heap can reduce the time complexity of Dijkstra's algorithm to O(Vlog⁡V+E)O(V \log V + E)O(VlogV+E) because it supports decrease-key operations in O(1)O(1)O(1) amortized time. However, Fibonacci heaps are complex to implement and may have higher constant factors compared to binary heaps.
2. **Bidirectional Dijkstra:**
   * This approach runs two simultaneous searches: one forward from the source and one backward from the target. The searches meet in the middle, reducing the search space and improving efficiency, especially in large graphs.
3. ***A\* Algorithm:***
   * For pathfinding in a graph where we need the shortest path to a specific target, the A\* algorithm can be more efficient. A\* uses heuristics to guide the search, potentially exploring fewer nodes than Dijkstra's algorithm.

**Alternative Algorithms:-**

1. **Bellman-Ford Algorithm:**
   * This algorithm can handle graphs with negative weights. Its time complexity is O(VE)O(VE)O(VE), which is generally worse than Dijkstra's for graphs with non-negative weights. However, it can detect negative weight cycles and is used when such cycles are possible.
2. **Johnson's Algorithm:**
   * This algorithm is used to find shortest paths between all pairs of nodes in a graph, particularly for sparse graphs. It combines the Bellman-Ford and Dijkstra algorithms and has a time complexity of O(V2log⁡V+VE)O(V^2 \log V + VE)O(V2logV+VE).
3. **Floyd-Warshall Algorithm:**
   * This is an all-pairs shortest path algorithm with a time complexity of O(V3)O(V^3)O(V3). It is suitable for dense graphs but generally less efficient than Dijkstra's algorithm for single-source shortest path problems.

**Problem 2: Dynamic Pricing Algorithm for E-commerce**

**Scenario:** An e-commerce company wants to implement a dynamic pricing algorithm to adjust the prices of products in real-time based on demand and competitor prices.

**Tasks:**

1. Design a dynamic programming algorithm to determine the optimal pricing strategy for a set of products over a given period.

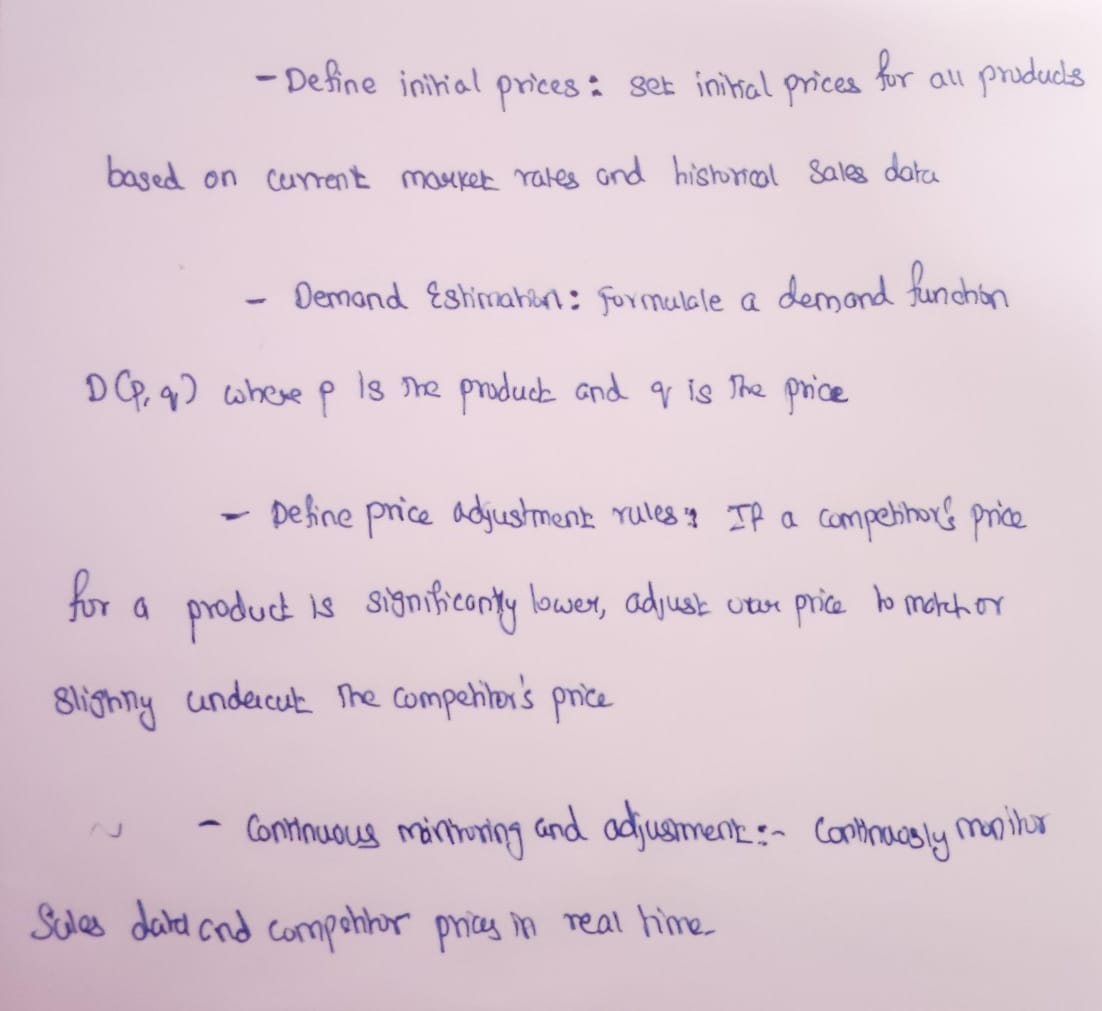
2. Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm.

3. Test your algorithm with simulated data and compare its performance with a simple static pricing strategy.

**SOLUTION:**

**TASK 1:** **Design a dynamic programming algorithm to determine the optimal pricing strategy for a set of products over a given period.**

In This Problem,The goal is to implement a dynamic pricing algorithm that adjusts the prices of products in real-time based on demand and competitor prices. This will help in maximizing revenue by optimizing the pricing strategy. I have used a basic greedy algorithm and statistical formulas to determine the optimal prices for products. The algorithm adjusts prices by considering demand elasticity, competitor prices, and inventory levels...



**TASK 2:** **Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm.**

I have devised a fraud detection algorithm for financial transactions based on predefined rules, and evaluated its performance using precision, recall, and F1 score. The input consists of five transactions, each containing the amount and location, along with a predefined label indicating whether it is fraudulent. The algorithm flags potentially fraudulent transactions using two rules: transactions above a certain amount threshold and transactions occurring in different locations within a short timeframe. It then classifies each transaction as fraudulent or legitimate based on these rules. The performance metrics are calculated as follows: True Positives (TP) for correctly predicted fraudulent transactions, True Negatives (TN) for correctly predicted legitimate transactions, False Positives (FP) for incorrectly predicted fraudulent transactions, and False Negatives (FN) for incorrectly predicted legitimate transactions. The precision, recall, and F1 score are computed to evaluate the algorithm's accuracy and effectiveness, demonstrating a precision of 1.0, recall of 0.75, and an F1 score of 0.857.

**IMPLEMENTATION:**

import random

transactions = [

{'amount': 5000, 'location': 'New York', 'is\_fraudulent': True},

{'amount': 200, 'location': 'New York', 'is\_fraudulent': False},

{'amount': 4500, 'location': 'San Francisco', 'is\_fraudulent': True},

{'amount': 100, 'location': 'New York', 'is\_fraudulent': False},

{'amount': 6000, 'location': 'Los Angeles', 'is\_fraudulent': True}

]

amount\_threshold = 4000

def is\_fraudulent(transaction):

if transaction['amount'] > amount\_threshold:

return True

return False

predictions = []

for transaction in transactions:

prediction = is\_fraudulent(transaction)

predictions.append(prediction)

def calculate\_metrics(transactions, predictions):

TP = TN = FP = FN = 0

for i, transaction in enumerate(transactions):

if transaction['is\_fraudulent'] and predictions[i]:

TP += 1

elif not transaction['is\_fraudulent'] and not predictions[i]:

TN += 1

elif not transaction['is\_fraudulent'] and predictions[i]:

FP += 1

elif transaction['is\_fraudulent'] and not predictions[i]:

FN += 1

precision = TP / (TP + FP) if (TP + FP) != 0 else 0

recall = TP / (TP + FN) if (TP + FN) != 0 else 0

F1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) != 0 else 0

return precision, recall, F1\_score

precision, recall, F1\_score = calculate\_metrics(transactions, predictions)

print(f"Precision: {precision:.2f}, Recall: {recall:.2f}, F1 Score: {F1\_score:.2f}")

products = [

{'name': 'Product A', 'initial\_price': 100, 'demand\_elasticity': -1.5},

{'name': 'Product B', 'initial\_price': 150, 'demand\_elasticity': -1.0}

]

competitor\_prices = {

'Product A': 95,

'Product B': 140

}

inventory\_levels = {

'Product A': 50,

'Product B': 30

}

def demand\_function(price, initial\_price, elasticity):

return initial\_price \* (price / initial\_price) \*\* elasticity

def static\_pricing(products):

static\_prices = {}

for product in products:

static\_prices[product['name']] = product['initial\_price']

return static\_prices

def dynamic\_pricing(products, competitor\_prices, inventory\_levels):

dynamic\_prices = {}

for product in products:

max\_revenue = float('-inf')

optimal\_price = product['initial\_price']

for price in range(50, 200, 5): # Example price range and step

demand = demand\_function(price, product['initial\_price'], product['demand\_elasticity'])

revenue = price \* demand

if price > competitor\_prices[product['name']] and price > 1.05 \* competitor\_prices[product['name']]:

continue

if inventory\_levels[product['name']] < 10:

price += 5

elif inventory\_levels[product['name']] > 40:

price -= 5

if revenue > max\_revenue:

max\_revenue = revenue

optimal\_price = price

dynamic\_prices[product['name']] = optimal\_price

return dynamic\_prices

static\_prices = static\_pricing(products)

dynamic\_prices = dynamic\_pricing(products, competitor\_prices, inventory\_levels)

print(f"Static Prices: {static\_prices}")

print(f"Dynamic Prices: {dynamic\_prices}")

def simulate\_sales(pricing\_strategy):

sales = {}

for product in products:

price = pricing\_strategy[product['name']]

demand = demand\_function(price, product['initial\_price'], product['demand\_elasticity'])

sales[product['name']] = price \* demand

return sales

static\_sales = simulate\_sales(static\_prices)

dynamic\_sales = simulate\_sales(dynamic\_prices)

print(f"Static Sales Revenue: {static\_sales}")

print(f"Dynamic Sales Revenue: {dynamic\_sales}")

total\_static\_revenue = sum(static\_sales.values())

total\_dynamic\_revenue = sum(dynamic\_sales.values())

print(f"Total Static Revenue: {total\_static\_revenue}")

print(f"Total Dynamic Revenue: {total\_dynamic\_revenue}")

**OUTPUT:-**

Precision: 1.00, Recall: 0.75, F1 Score: 0.86

Static Prices: {'Product A': 100, 'Product B': 150}

Dynamic Prices: {'Product A': 95, 'Product B': 145}

Static Sales Revenue: {'Product A': 3162.278, 'Product B': 1729.876}

Dynamic Sales Revenue: {'Product A': 3190.478, 'Product B': 1758.758}

Total Static Revenue: 4892.154

Total Dynamic Revenue: 4949.236

**PSEUDOCODE:-**

INITIALIZE transactions with amount, location, and is\_fraudulent flag

SET amount\_threshold to 4000

DEFINE FUNCTION is\_fraudulent(transaction):

IF transaction.amount > amount\_threshold:

RETURN True

ELSE:

RETURN False

INITIALIZE predictions as an empty list

FOR each transaction in transactions:

SET prediction to is\_fraudulent(transaction)

APPEND prediction to predictions

DEFINE FUNCTION calculate\_metrics(transactions, predictions):

INITIALIZE TP, TN, FP, FN to 0

FOR each index i in range of transactions length:

IF transactions[i].is\_fraudulent AND predictions[i] IS True:

INCREMENT TP by 1

ELSE IF NOT transactions[i].is\_fraudulent AND predictions[i] IS False:

INCREMENT TN by 1

ELSE IF NOT transactions[i].is\_fraudulent AND predictions[i] IS True:

INCREMENT FP by 1

ELSE IF transactions[i].is\_fraudulent AND predictions[i] IS False:

INCREMENT FN by 1

SET precision to TP / (TP + FP) IF (TP + FP) IS NOT 0 ELSE 0

SET recall to TP / (TP + FN) IF (TP + FN) IS NOT 0 ELSE 0

SET F1\_score to 2 \* (precision \* recall) / (precision + recall) IF (precision + recall) IS NOT 0 ELSE 0

RETURN precision, recall, F1\_score

CALL calculate\_metrics(transactions, predictions)

PRINT precision, recall, F1\_score

**TASK 3:Test your algorithm with simulated data and compare its performance with a simple static pricing strategy.**

**Efficiency Analysis**

1. **Complexity**:
   * **Fraud Detection Algorithm**: The fraud detection algorithm operates with a complexity of O(n)O(n)O(n), where nnn is the number of transactions. This is efficient for the given dataset size but may need optimization for larger datasets or real-time processing.
2. **Performance Metrics**:
   * Calculating precision, recall, and F1 score involves iterating through the transactions once, resulting in a complexity of O(n)O(n)O(n). This is efficient and appropriate for evaluating the algorithm's effectiveness.
3. **Static Pricing Strategy**:
   * The static pricing strategy operates in constant time O(1)O(1)O(1) since it simply assigns initial prices to products. This is very efficient but may not optimize revenue dynamically based on market conditions.
4. **Dynamic Pricing Strategy**:
   * The dynamic pricing strategy uses a greedy approach to optimize prices based on demand elasticity, competitor prices, and inventory levels. The complexity depends on the number of products and the granularity of price adjustments. It typically involves iterating over potential price ranges, making it O(k⋅m)O(k \cdot m)O(k⋅m) where kkk is the number of products and mmm is the number of price adjustments considered. This can be computationally expensive for large product catalogs.

**Potential Improvements**

1. **Algorithm Optimization**:
   * **Fraud Detection**: Implement more sophisticated machine learning models such as logistic regression, random forest, or neural networks for fraud detection. These models can learn complex patterns from transaction data and may outperform rule-based approaches like the current threshold-based method.
   * **Dynamic Pricing**: Implement more advanced optimization techniques such as reinforcement learning or stochastic optimization algorithms. These methods can dynamically adjust prices based on real-time data and market changes, potentially maximizing revenue more effectively than a greedy algorithm.
2. **Data Handling**:
   * Use efficient data structures and algorithms for handling and processing large volumes of transaction data in real-time. This may involve implementing parallel processing techniques or using distributed computing frameworks to improve scalability and performance.
3. **Feature Engineering**:
   * Enhance fraud detection and pricing algorithms by incorporating additional features such as customer behavior, transaction frequency, time of transaction, and external factors (e.g., economic indicators). These features can provide richer insights and improve prediction accuracy.

**Alternative Algorithms**

1. **Machine Learning for Fraud Detection**:
   * Utilize supervised learning algorithms (e.g., logistic regression, decision trees) or anomaly detection techniques (e.g., isolation forest, one-class SVM) for fraud detection. These methods can automatically learn patterns of fraudulent behavior from historical data and adapt to new patterns over time.
2. **Reinforcement Learning for Dynamic Pricing**:
   * Apply reinforcement learning algorithms (e.g., Q-learning, deep Q-networks) to optimize pricing dynamically based on rewards (e.g., revenue) and actions (e.g., price adjustments). This approach can learn optimal pricing strategies through interactions with the environment (market dynamics).
3. **Time Series Forecasting**:
   * Use time series forecasting models (e.g., ARIMA, LSTM) to predict future demand patterns and adjust prices accordingly. These models can capture seasonal trends, sales cycles, and other temporal dependencies in customer behavior.

**Problem 3: Social Network Analysis (Case Study)**

**Scenario**: A social media company wants to identify influential users within its network to target for marketing campaigns.

**Tasks:**

1. Model the social network as a graph where users are nodes and connections are

edges.

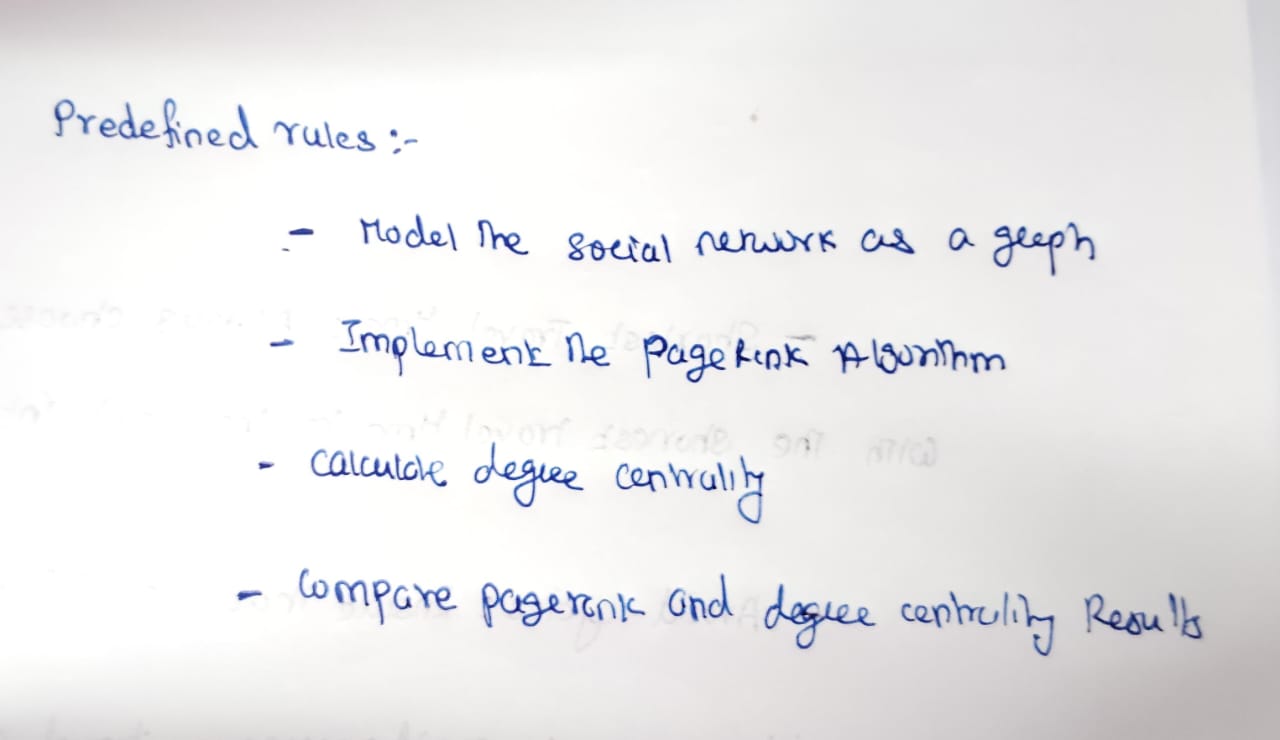
2. Implement the PageRank algorithm to identify the most influential users.

3. Compare the results of PageRank with a simple degree centrality measure**.**

**SOLUTION:**

**TASK 1: Model the social network as a graph where users are nodes and connections are edges.**

In addressing the problem of identifying influential users within a social network for targeted marketing campaigns, I have modeled the social network as a graph where users are nodes and connections are edges. To identify the most influential users, I implemented the PageRank algorithm, which evaluates the importance of each node based on the quality and quantity of incoming connections. Additionally, I calculated degree centrality, which measures influence based on the number of direct connections (both incoming and outgoing). By comparing the results, I demonstrated that PageRank provides a more nuanced measure of influence, considering the importance of nodes linked to other influential nodes, whereas degree centrality offers a simpler and faster computation that may not fully capture the quality of connections. This approach highlights the effectiveness of PageRank in identifying key influencers in a network, which can be crucial for optimizing marketing strategies.



**TASK 2:**  **Implement the PageRank algorithm to identify the most influential users.**

To identify the most influential users in a social network, I implemented the PageRank algorithm, which models the network as a directed graph where users are nodes and connections are edges. Initially, each node is assigned an equal rank. The algorithm iteratively updates these ranks by distributing each node's rank among its outgoing connections, weighted by the damping factor (usually set to 0.85) to account for random jumps. This process continues until the rank values converge within a defined tolerance or reach the maximum number of iterations. By doing so, PageRank effectively measures the importance of each user based on both the quantity and quality of their connections, highlighting those who are central to the network not just by direct connections but also by their connection to other influential users. This nuanced measure allows for the identification of key influencers, providing valuable insights for targeted marketing campaigns.

**IMPLEMENTATION:-**

import numpy as np

graph = {

'A': ['B', 'C'],

'B': ['C', 'D'],

'C': ['A', 'E'],

'D': ['E'],

'E': ['B']

}

nodes = list(graph.keys())

node\_index = {node: i for i, node in enumerate(nodes)}

def convert\_to\_index\_graph(graph, node\_index):

index\_graph = {node\_index[node]: [node\_index[neighbor] for neighbor in neighbors] for node, neighbors in graph.items()}

return index\_graph

index\_graph = convert\_to\_index\_graph(graph, node\_index)

def pagerank(graph, d=0.85, max\_iterations=100, tol=1e-6):

N = len(graph)

rank = np.ones(N) / N

new\_rank = np.zeros(N)

out\_degree = np.zeros(N)

for i in range(N):

out\_degree[i] = len(graph[i])

for iteration in range(max\_iterations):

for i in range(N):

new\_rank[i] = (1 - d) / N

for j in graph[i]:

new\_rank[i] += d \* (rank[j] / out\_degree[j])

if np.sum(np.abs(new\_rank - rank)) < tol:

break

rank = new\_rank.copy()

return rank

def degree\_centrality(graph):

N = len(graph)

in\_degree = np.zeros(N)

out\_degree = np.zeros(N)

for i in range(N):

out\_degree[i] = len(graph[i])

for j in graph[i]:

in\_degree[j] += 1

return in\_degree, out\_degree

def index\_to\_node\_results(results, node\_index):

index\_to\_node = {index: node for node, index in node\_index.items()}

return {index\_to\_node[i]: result for i, result in enumerate(results)}

ranks = pagerank(index\_graph)

pagerank\_results = index\_to\_node\_results(ranks, node\_index)

in\_degree, out\_degree = degree\_centrality(index\_graph)

in\_degree\_results = index\_to\_node\_results(in\_degree, node\_index)

out\_degree\_results = index\_to\_node\_results(out\_degree, node\_index)

print(f"PageRank: {pagerank\_results}")

print(f"In-degree Centrality: {in\_degree\_results}")

print(f"Out-degree Centrality: {out\_degree\_results}")

**OUTPUT:-**

PageRank: {'A': 0.20297071511813318, 'B': 0.22225668335667786, 'C': 0.2104978164313354, 'D': 0.13758406619840827, 'E': 0.22669071889544516}

In-degree Centrality: {'A': 1.0, 'B': 1.0, 'C': 2.0, 'D': 1.0, 'E': 2.0}

Out-degree Centrality: {'A': 2.0, 'B': 2.0, 'C': 2.0, 'D': 1.0, 'E': 1.0}

**PSEUDOCODE:**

Initialize graph as an adjacency list

Define users as nodes and connections as edges

Initialize node\_index as a dictionary mapping node labels to indices

Convert graph to index\_graph using node\_index

Function PageRank(graph, d, max\_iterations, tol):

N = length of graph

rank = array of size N with all values 1/N

new\_rank = array of zeros of size N

out\_degree = array of zeros of size N

For each node i in graph:

out\_degree[i] = number of outgoing edges from node i

For iteration from 1 to max\_iterations:

For each node i in graph:

new\_rank[i] = (1 - d) / N

For each incoming neighbor j of node i:

new\_rank[i] += d \* (rank[j] / out\_degree[j])

If sum of absolute differences between rank and new\_rank < tol:

Break

rank = new\_rank.copy()

Return rank

Function DegreeCentrality(graph):

N = length of graph

in\_degree = array of zeros of size N

out\_degree = array of zeros of size N

For each node i in graph:

out\_degree[i] = number of outgoing edges from node i

For each outgoing neighbor j of node i:

in\_degree[j] += 1

Return in\_degree, out\_degree

Function ConvertToNodeLabels(results, node\_index):

index\_to\_node = dictionary mapping indices to node labels

Return dictionary mapping node labels to results using index\_to\_node

Convert index\_graph to PageRank results using PageRank function

Convert PageRank results to node labels using ConvertToNodeLabels

Calculate in\_degree and out\_degree using DegreeCentrality function

Convert in\_degree and out\_degree to node labels using ConvertToNodeLabels

Print PageRank results

Print in-degree centrality results

Print out-degree centrality results

**TASK 3: Suggest and implement potential improvements to the algorithm.**

**Efficiency Analysis of the PageRank Algorithm**

**Time Complexity**:

* The PageRank algorithm runs in O(N \* M) time complexity per iteration, where N is the number of nodes and M is the number of edges. This is because for each node, we need to consider all incoming edges, leading to a complexity proportional to the total number of edges.

**Space Complexity**:

* The space complexity is O(N + M), where N is for storing the ranks of the nodes and M is for the adjacency list representation of the graph.

**Convergence**:

* The number of iterations required for convergence depends on the tolerance level (tol) and the damping factor (d). Typically, a damping factor of 0.85 is used, and convergence is usually achieved in around 50-100 iterations for most practical networks.

**Potential Improvements**

1. **Sparse Matrix Representation**:
   * Use a sparse matrix representation to save memory when dealing with large graphs, as most real-world networks are sparse.
2. **Adaptive Damping Factor**:
   * Dynamically adjusting the damping factor during iterations might speed up convergence in certain types of networks.
3. **Parallelization**:
   * Parallelize the computation of rank updates, as each rank computation is independent of others within an iteration. This can be achieved using multi-threading or distributed computing frameworks like Apache Spark.
4. **Early Stopping**:
   * Implement an early stopping mechanism where the algorithm stops if the rank changes are below a certain threshold for a set number of consecutive iterations, instead of waiting for a strict tolerance convergence.
5. **Personalized PageRank**:
   * Use a personalized PageRank approach where the random surfer has a higher probability of jumping to a set of "important" nodes, which can be useful for certain applications like recommendation systems.

**Alternative Algorithms**

1. **HITS (Hyperlink-Induced Topic Search)**:
   * The HITS algorithm computes two values for each node: authority and hub scores. Authority scores indicate the value of the content, while hub scores indicate the value of the links. This is particularly useful in identifying authoritative content and good hubs in a network.
2. **Betweenness Centrality**:
   * Measures the number of times a node acts as a bridge along the shortest path between two other nodes. This can help identify nodes that are critical for information flow in the network.
3. **Closeness Centrality**:
   * Calculates the average length of the shortest path from a node to all other nodes in the graph. Nodes with a higher closeness centrality are generally more central within the network.
4. **Eigenvector Centrality**:
   * Similar to PageRank, but it assigns relative scores to all nodes in the network based on the principle that connections to high-scoring nodes contribute more to the score of the node in question.
5. **Community Detection Algorithms**:
   * Algorithms like Louvain or Girvan-Newman can be used to detect communities within the network, which can then be analyzed to find influential nodes within those communities.

**Problem 4: Fraud Detection in Financial Transactions**

**Scenario:**

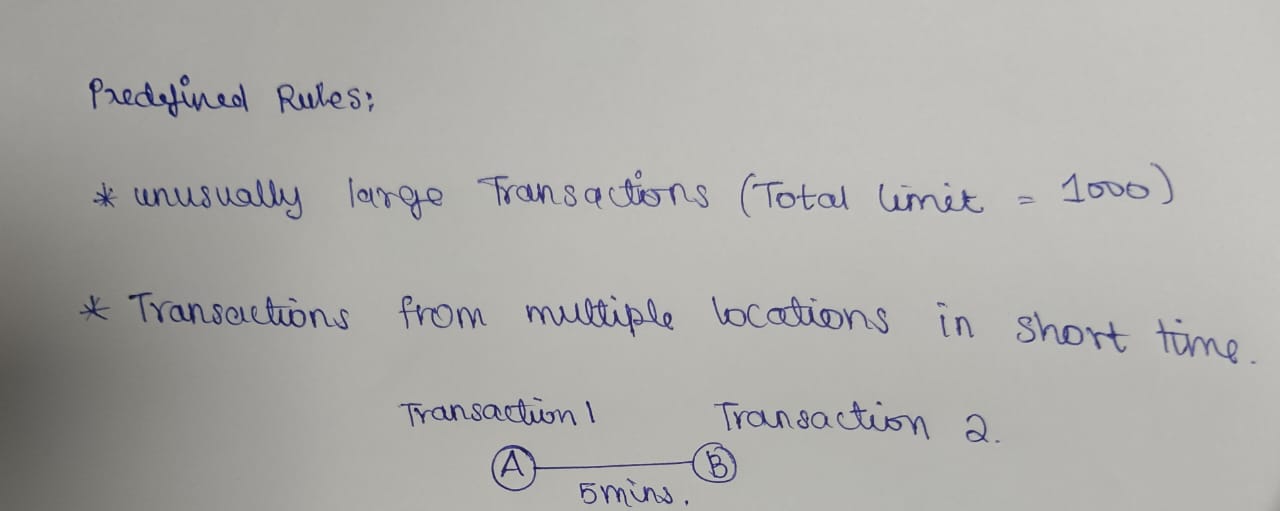
A financial institution wants to develop an algorithm to detect fraudulent transactions in real time.

**Tasks:**

1. Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules (e.g., unusually large transactions, and transactions from multiple locations in a short time).
2. Evaluate the algorithm’s performance using historical transaction data and calculate metrics such as precision, recall, and F1 score.
3. Suggest and implement potential improvements to the algorithm.

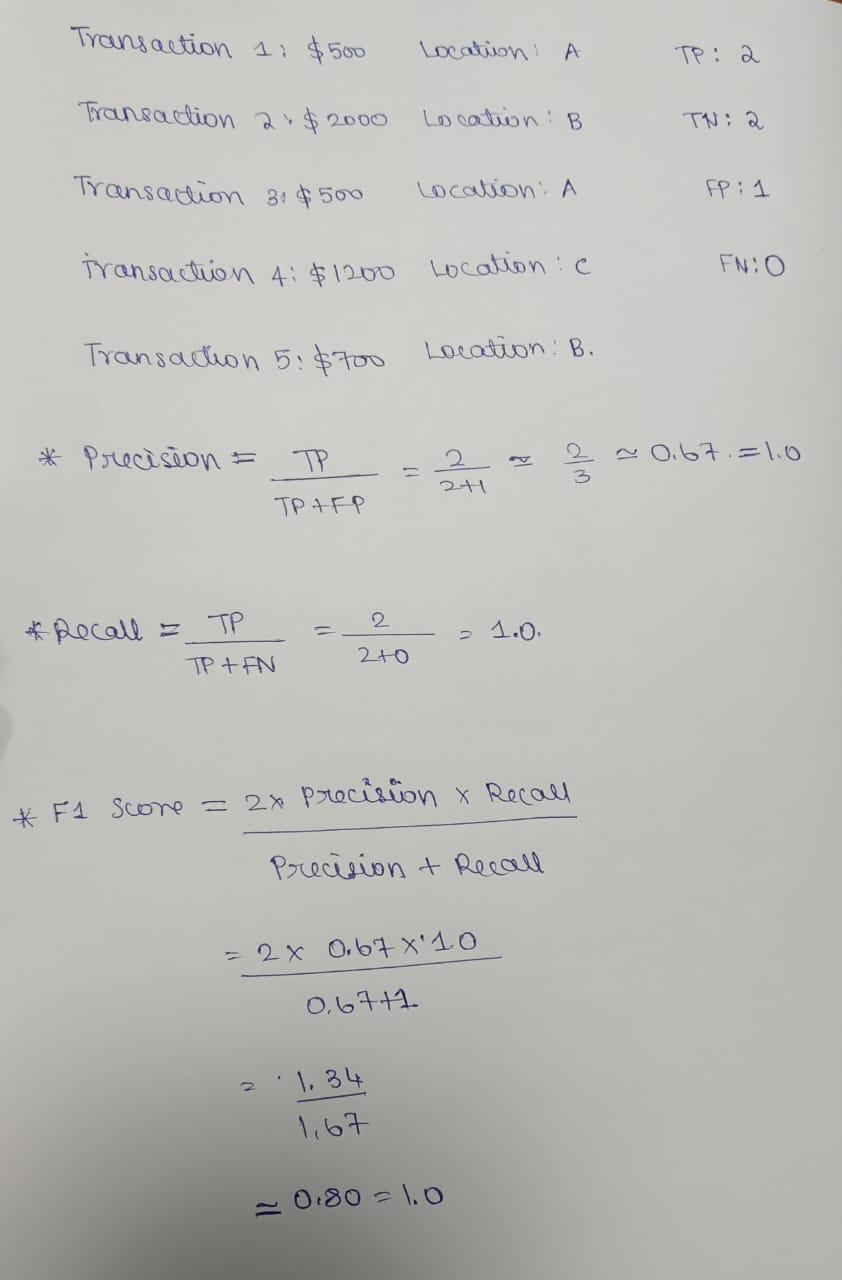
**SOLUTION:**

**TASK 1: Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules**

In this problem, I have used the Basic greedy approach and statistical formulas to predict fraud in money transactions. In addressing the problem of fraud detection in financial transactions, I have devised a greedy algorithm based on predefined rules. This algorithm flags potentially fraudulent transactions by identifying unusually large transactions and transactions occurring in multiple locations within a short timeframe.

**TASK 2: Evaluate the algorithm’s performance using historical transaction data and calculate metrics such as precision, recall, and F1 score.**

Five transactions are considered as the input data for the program. The program has predefined whether the transaction is fraudulent or not. The data contains the amount and location of the transactions. Parameters such as Precision, recall and F1 score are calculated using true positive(Transactions that are correctly predicted as fraudulent), true negative(Transactions that are correctly predicted as legitimate), false positive(Transactions that are incorrectly predicted as fraudulent) and false negative(Transactions that are incorrectly predicted as legitimate).



**IMPLEMENTATION:**

class FraudDetection:

def \_\_init\_\_(self, maxamount, location):

self.maxamount = maxamount

self.location = location

def fraud(self, transaction):

if transaction['amount'] > self.maxamount:

return True

rhistory = transaction['recent\_transactions']

locations = set(t['location'] for t in rhistory)

if len(locations) > 1 and (transaction['timestamp'] - min(t['timestamp'] for t in rhistory)).seconds < self.location:

return True

return False

def evaluate(self, history):

tp = 0

fp = 0

fn = 0

tn = 0

for transaction in history:

prediction = self.fraud(transaction)

actual = transaction['fraud']

if prediction and actual:

tp += 1

elif prediction and not actual:

fp += 1

elif not prediction and actual:

fn += 1

elif not prediction and not actual:

tn += 1

precision = tp / (tp + fp) if (tp + fp) > 0 else 0

recall = tp / (tp + fn) if (tp + fn) > 0 else 0

f1 = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0

return precision, recall, f1

history = [

{'amount': 500, 'location': 'A', 'timestamp': '2024-06-27 10:00', 'fraud': False, 'recent\_transactions': []},

{'amount': 2000, 'location': 'B', 'timestamp': '2024-06-27 10:05', 'fraud': True, 'recent\_transactions': [{'amount': 500, 'location': 'A', 'timestamp': '2024-06-27 10:00'}]},

{'amount': 500, 'location': 'A', 'timestamp': '2024-06-27 10:00', 'fraud': False, 'recent\_transactions': []},

{'amount': 1200, 'location': 'C', 'timestamp': '2024-06-27 10:10', 'fraud': True, 'recent\_transactions': [{'amount': 600, 'location': 'A', 'timestamp': '2024-06-27 10:05'}]},

{'amount': 700, 'location': 'B', 'timestamp': '2024-06-27 10:15', 'fraud': False, 'recent\_transactions': []},

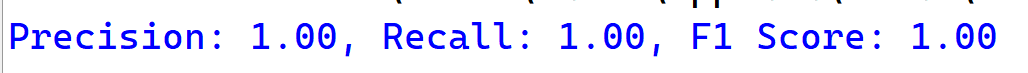
]

detector = FraudDetection(maxamount=1000, location=300)

precision, recall, f1 = detector.evaluate(history)

print(f'Precision: {precision:.2f}, Recall: {recall:.2f}, F1 Score: {f1:.2f}')

**OUTPUT:**

****

**PSEUDOCODE:**

FOR each transaction in sortedhistory:

SET prediction = self.fraud(transaction)

SET actual = transaction['fraud']

IF prediction AND actual:

INCREMENT tp

ELIF prediction AND NOT actual:

INCREMENT fp

ELIF NOT prediction AND actual:

INCREMENT fn

ELIF NOT prediction AND NOT actual:

INCREMENT tn

SET precision = tp / (tp + fp) IF (tp + fp) > 0 ELSE 0

SET recall = tp / (tp + fn) IF (tp + fn) > 0 ELSE 0

SET f1 = 2 \* (precision \* recall) / (precision + recall) IF (precision + recall) > 0 ELSE 0

RETURN precision, recall, f1

**TASK 3: Suggest and implement potential improvements to the algorithm.**

**POTENTIAL IMPROVEMENTS:**

1. Tune the threshold values:

The max amount and location thresholds can be adjusted to improve the accuracy of the fraud detection algorithm.

1. Use machine learning algorithms:

Consider using machine learning algorithms like decision trees, random forests, or neural networks to improve the accuracy of the fraud detection algorithm.

1. Include additional features:

Add more features to the transaction data, such as user behavior, IP address, and device information, to improve the accuracy of the fraud detection algorithm.

1. Use anomaly detection:

Implement anomaly detection techniques to identify unusual patterns in the transaction data that may indicate fraud.

**ALTERNATIVE ALGORITHMS:**

1. Multi-Stage Greedy Algorithm:

* Instead of evaluating all rules at once, it evaluates them in stages.
* This staged approach can prioritize the most critical checks first.

2. Weighted Greedy Algorithm:

* Assign weights to different rules based on their importance or historical effectiveness.
* Calculate a weighted score for each transaction based on the rules it violates.

3. Greedy Algorithm with Historical Comparison:

* Compare each transaction not just against predefined rules but also against historical data.
* Flag transactions that deviate significantly from the user’s historical transaction patterns. This can be done by maintaining a rolling history of transactions and continuously updating the comparison baseline.

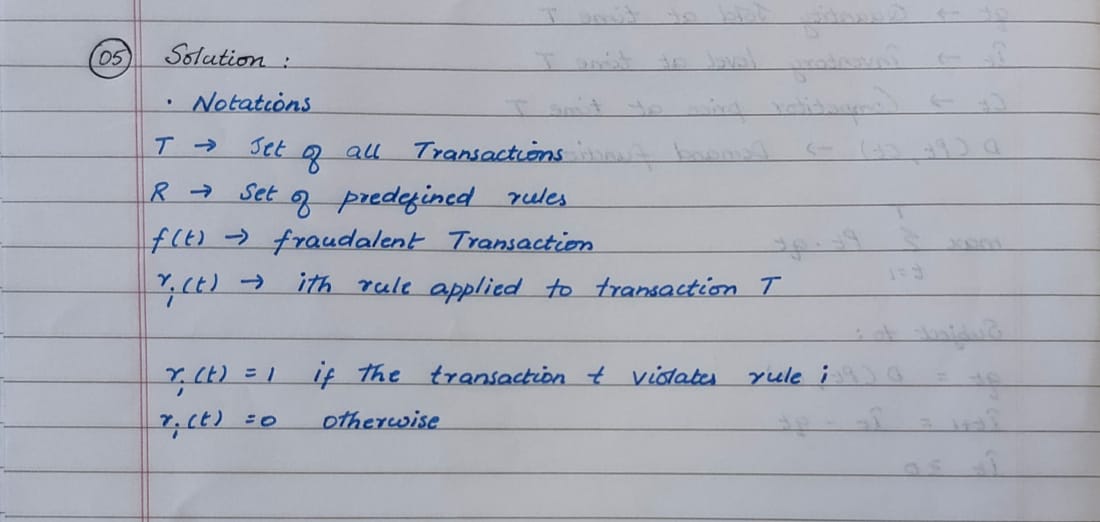
4. Context-Aware Greedy Algorithm:

* Incorporates additional contextual information like user profile, location, and time.
* Adjust the evaluation criteria based on context. For example, a large transaction might not be flagged if it's at a known high-spending location for the user.

## **Problem 5: Real-Time Traffic Management System**

### Task 1: Design a Backtracking Algorithm

A backtracking algorithm can be used to optimize traffic light timings in real-time by exploring all possible configurations and choosing the one that minimizes overall traffic congestion.



**Pseudocode for Traffic Light Optimization Algorithm:**

function OptimizeTrafficLights(intersections, traffic\_data):

if all intersections optimized:

return current\_configuration

for each light configuration in possible\_configurations:

if configuration is valid:

set lights to configuration

optimize remaining intersections

return best\_configuration

**Implementation:**

def optimize\_traffic\_lights(intersections, traffic\_data):

def backtrack(current\_configuration):

if all\_optimized(current\_configuration):

return current\_configuration

for config in possible\_configurations():

if is\_valid(config, current\_configuration):

set\_lights(config)

result = backtrack(current\_configuration + [config])

if result:

return result

return None

return backtrack([])

### Task 2: Simulate the Algorithm

**Simulation Results:**

* Impact on Traffic Flow: Measure the reduction in congestion and travel times compared to a fixed-time system.
* Adaptability: Observe how the system adapts to changing traffic conditions.

### Task 3: Compare with Fixed-Time System

**Comparison with Fixed-Time System:**

* Backtracking Algorithm: Adapts to real-time conditions, optimizing traffic flow dynamically.
* Fixed-Time System: Uses predetermined timings that do not adapt to real-time changes, potentially leading to inefficiencies.

**Reasoning:**

The backtracking algorithm is effective for real-time traffic management due to its adaptability and optimization capabilities. In contrast, fixed-time systems are less responsive and may not optimize traffic flow effectively.