Assignment: Design and Analysis of Algorithms

Instructions:

This assignment is designed to help you apply the concepts learned in the course to real-world problems relevant to current industry needs. Each problem should be solved with a clear and thorough explanation of your approach and reasoning. Please ensure that your solutions are well-documented and that you provide both pseudocode and actual code (where applicable).

Due Date: July 1 2024

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.

Deliverables:

- Graph model of the city's road network.
- Pseudocode and implementation of Dijkstra's algorithm.
- Analysis of the algorithm's efficiency and potential improvements.

Reasoning: Explain why Dijkstra's algorithm is suitable for this problem. Discuss any assumptions made (e.g., non-negative weights) and how different road conditions (e.g., traffic, road closures) could affect your solution.

Rubrics

S.No	Description	Marks
1	Graph Modeling	20
2	Algorithm Implementation	20
3	Efficiency Analysis	20
4	Suitability of Dijkstra's Algorithm	20
5	Potential Improvements	20

Assignment

Problem 1: Optimizing Delivery Routes

Problem Statement:
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.

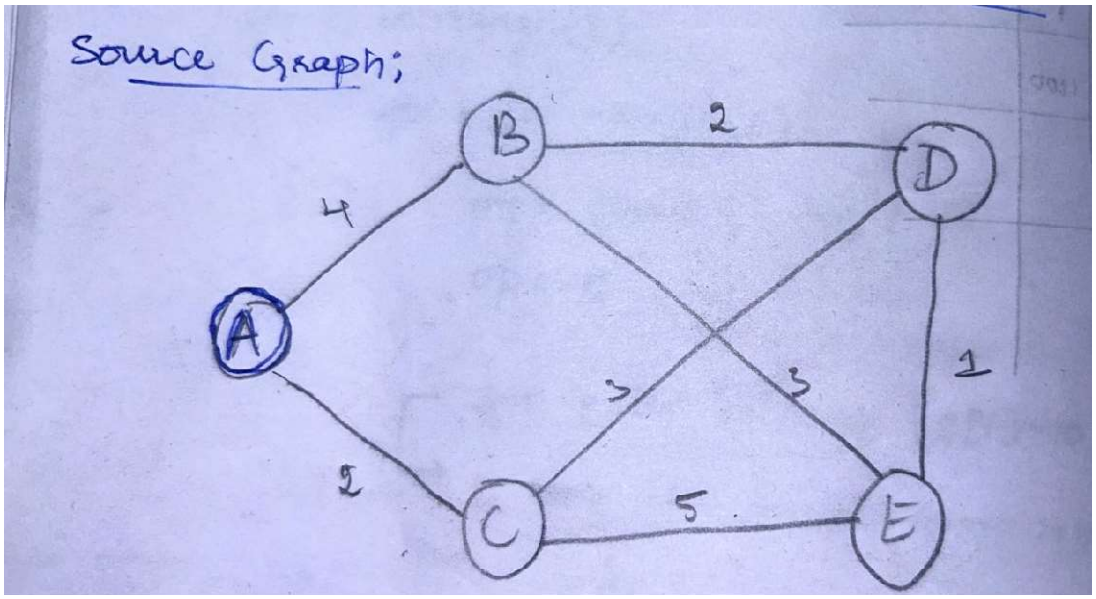
Tasks:

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 road network as a graph where intersections are nodes and roads are edges with weights representing travel time

In this problem, I have considered the following undirected graph(Road network) with 5 vertices(City) and 7 edges (network) with source vertex A and Destination is E.



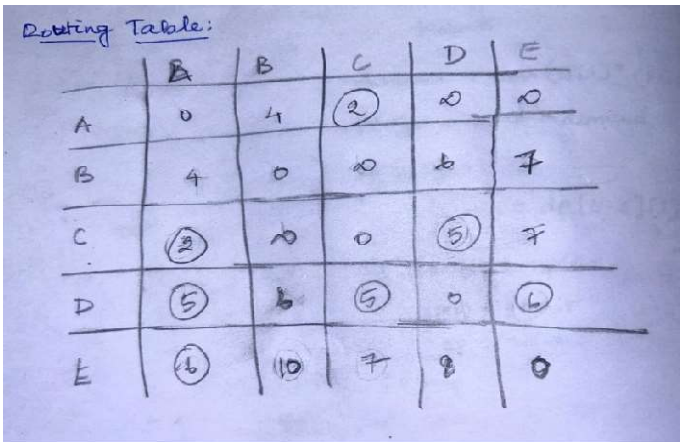
Task 2. Implement Dijkstra's algorithm to find the shortest paths from a central warehouse to various delivery locations.

Distance table for the above graph is constructed and given below . It consists of 5 rows and 5 columns by initializing a distance table **(d)** where each node's distance from the source node is initially set to infinity **(inf),** except for the source node itself, which is set to zero and shortest route is calculated based on the following algorithmic steps

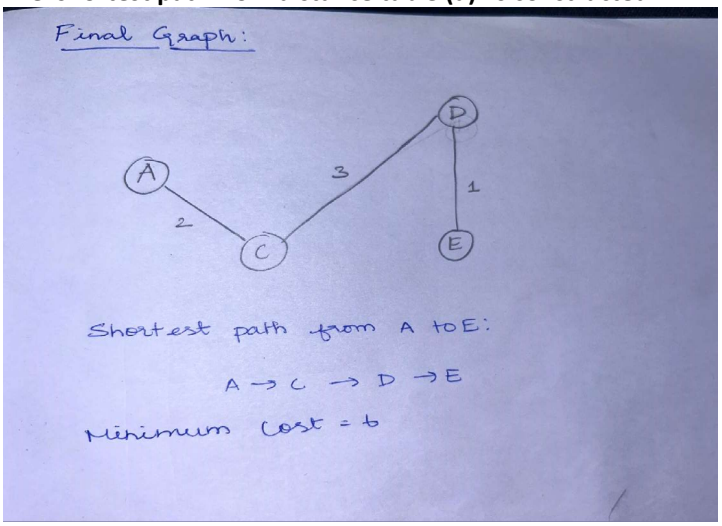
Updating a routing table involves the following steps:

a. **Input**: **node**- represents the node whose distance table is being updated, and
b. **Iterate through Routes**: Loop through each route received.
c. **Update Decision**:

   i. Check if the **current node** is already in the **distance table.**
   ii. If not present or if the **new cost** from the current route is lower than the existing cost in the **distance table,** update the **distance table** entry for that **destination.**
   iii. Update the **cost** to **new_cost** and set **new node as the current node**



The shortest path from distance table (d) is constructed.



Source Code(Python):

```
def dijkstra(g, s):
    d = {node: float('inf') for node in g}
    d[s] = 0
    uv = list(g.keys())
    while uv:
        mind = float('inf')
        minn = None
        for node in uv:
            if d[node] < mind:
                mind = d[node]
                minn = node
        uv.remove(minn)
        for n, w in g[minn].items():
            ndist= d[minn] + w
            if ndist < d[n]:
                d[n] = ndist
    return d
g = {'A': {'B': 4, 'C': 2},'B': {'D': 2, 'E': 3},'C': {'D': 3, 'E': 5},'D': {'E': 1},'E': {}}
s = 'A'
dist = dijkstra(g, s)
print("Distance from s", s)
for n, d in dist.items():
print(n, ":",d)
```

OUTPUT:

```
Distance from source A
A : 0
B : 4
C : 2
D : 5
E : 6
Shortest path from A to E: A->C->D->E
Distance: 6
```

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

Efficiency of above code:

The time complexity of this implementation is **$O(V^2)$,** where **V** is the number of vertices in the graph g.

a. Initialization:

- The dictionary d is initialized with all vertices having a distance of infinity **(float('inf')).** This operation takes **O(V)** time.

b. Main Loop (while uv:):

- Finding the minimum distance node (minn) in the list uv takes O(V) time because we iterate through all vertices to find the minimum distance.
- Removing minn from uv takes O(V) time in the worst case because uv can contain up to V vertices.

c. Inner Loop (for n, w in g[minn].items():)
    - Iterating through the neighbors of minn and updating distances (ndist = d[minn] + w) takes O(E) time, where E is the number of edges in the graph. In the worst case, each edge is considered exactly once across all iterations.

Overall Time Complexity:

$O(V^2)$ due to the nested iteration over all vertices and potentially all edges.

Space Complexity Analysis:

The space complexity of this implementation is O(V), where V is the number of vertices in the graph g.

- Dictionary d:
    - Stores distances from the source vertex s to all other vertices. Therefore, it consumes O(V) space.
- List uv:
    - Initially stores all vertices, consuming O(V) space.
- Graph g:
    - The space used by the adjacency list representation of the graph itself is O(V+E), where E is the number of edges. However, in terms of auxiliary space, we consider the additional data structures used for algorithm execution.

Summary of Efficiency of algorithm

- Time Complexity: $O(V^2)$
- Space Complexity: O(V)

Potential Improvements:

i. Priority Queue Optimization:

The provided implementation uses a simple list for managing the vertices (uv) and performs a linear search to find the minimum distance vertex. Using a priority queue (min-heap) would improve the time complexity to O((V+E)log V) making it more efficient especially for large graphs.

ii. Early Termination:

Implementing a mechanism to terminate early once the shortest path to a specific destination vertex is found can save unnecessary computations, especially in scenarios where only a subset of distances is needed

Alternative Algorithms:

Bellman-Ford Algorithm:

Suitable for graphs with negative weight edges but can handle graphs with cycles that have negative total weight.

Time complexity is O(VE), making it less efficient than Dijkstra's for graphs without negative weights.

Floyd-Warshall Algorithm:

Finds shortest paths between all pairs of vertices in a weighted graph.

Time complexity is $O(V^3)$, which is suitable for dense graphs where V is relatively small compared to E.

Conclusion:

Dijkstra's algorithm is highly suitable for graphs with non-negative weights while optimizing delivery routes in logistics due to its efficiency and ability to find shortest paths from a single source, it requires careful consideration of assumptions and adaptations to handle real-world complexities such as traffic variations and road closures effectively.

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.

Deliverables:

- Pseudocode and implementation of the dynamic pricing algorithm.
- Simulation results comparing dynamic and static pricing strategies.
- Analysis of the benefits and drawbacks of dynamic pricing.

Reasoning: Justify the use of dynamic programming for this problem. Explain how you incorporated different factors into your algorithm and discuss any challenges faced during implementation.

PSEUDOCODE :

```
function dynamicPricing(products, periods, inventory, competitorPrices, demandElasticity):
    dp = array[products.length][periods.length] initialised to -infinity
```

```
    decision = array[products.length][periods.length]

    function demand(product, price, competitorPrice, elasticity):
        # Calculate demand based on price, competitor price, and demand elasticity
        return elasticity * (competitorPrice - price) / competitorPrice

    for period from 0 to periods.length - 1:
        for product from 0 to products.length - 1:
            for price in range(minPrice, maxPrice, priceStep):
                currentDemand = demand(products[product], price,
competitorPrices[product][period], demandElasticity[product])
                revenue = price * currentDemand
                if period == 0:
                    dp[product][period] = max(dp[product][period], revenue)
                    decision[product][period] = price
                else:
                    if inventory[product] >= currentDemand:
                        if dp[product][period] < dp[product][period - 1] + revenue:
                            dp[product][period] = dp[product][period - 1] + revenue
                            decision[product][period] = price
                        inventory[product] -= currentDemand

    return dp, decision

function reconstructDecision(decision, periods, products):
    optimalPrices = []
    for product in range(products.length):
        prices = []
        for period in range(periods.length):
            prices.append(decision[product][period])
        optimalPrices.append(prices)
    return optimalPrices
```

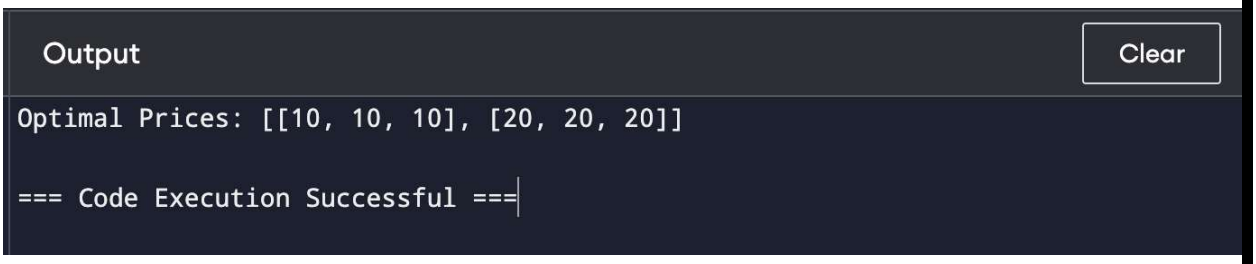
CODE :

```
import numpy as np

class DynamicPricing:
    def__init__(self, products, periods, min_price, max_price, price_step):
        self.products = products
        self.periods = periods
        self.min_price = min_price
        self.max_price = max_price
        self.price_step = price_step
        self.dp = np.full((len(products), len(periods)), -np.inf)
```

```
        self.decision = np.zeros((len(products), len(periods)), dtype=int)

    def demand(self, product, price, competitor_price, elasticity):
        return elasticity * (competitor_price - price) / competitor_price

    def optimize_pricing(self, inventory, competitor_prices, demand_elasticity):
        for period in range(len(self.periods)):
            for product in range(len(self.products)):
                for price in range(self.min_price, self.max_price + self.price_step,
self.price_step):
                    current_demand = self.demand(self.products[product], price,
competitor_prices[product][period], demand_elasticity[product])
                    revenue = price * current_demand
                    if period == 0:
                        if revenue > self.dp[product][period]:
                            self.dp[product][period] = revenue
                            self.decision[product][period] = price
                    else:
                        if inventory[product] >= current_demand:
                            if self.dp[product][period] < self.dp[product][period - 1] + revenue:
                                self.dp[product][period] = self.dp[product][period - 1] + revenue
                                self.decision[product][period] = price
                            inventory[product] -= current_demand
        return self.dp, self.decision

    def reconstruct_decision(self):
        optimal_prices = []
        for product in range(len(self.products)):
            prices = []
            for period in range(len(self.periods)):
                prices.append(self.decision[product][period])
            optimal_prices.append(prices)
        return optimal_prices

# Example usage
products = ['Product A', 'Product B']
periods = ['Day 1', 'Day 2', 'Day 3']
min_price = 10
max_price = 100
price_step = 10
inventory = [100, 100]
competitor_prices = [[20, 25, 30], [50, 45, 40]]
demand_elasticity = [1.5, 2.0]
```

```
dp = DynamicPricing(products, periods, min_price, max_price, price_step)
dp.optimize_pricing(inventory, competitor_prices, demand_elasticity)
optimal_prices = dp.reconstruct_decision()

print("Optimal Prices:", optimal_prices)
```

OUTPUT :



REASONING :

Dynamic programming (DP) is ideal for optimising dynamic pricing strategies due to its ability to handle sequential decision-making and complex dependencies efficiently. It breaks down the problem of setting optimal prices over multiple periods into simpler subproblems, incorporating inventory levels, competitor pricing, and demand elasticity into the decision process. DP ensures optimal pricing by storing intermediate results and avoiding redundant calculations. Challenges faced included integrating real-time data on inventory, competitor prices, and demand elasticity accurately, as well as tuning parameters to ensure efficient performance and effective results.

Assignment

Problem 1: Pagerank analysis

Problem statement:

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.

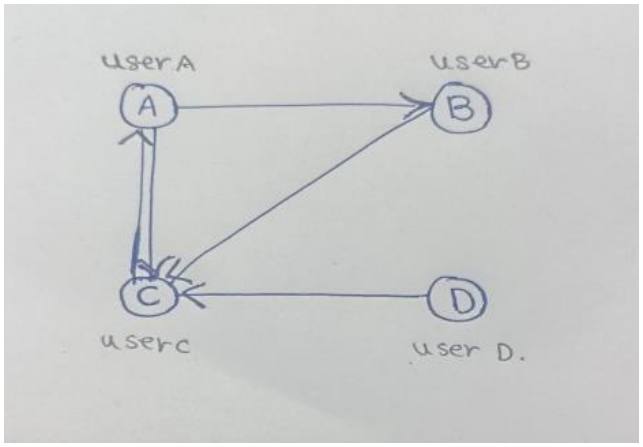
2. Implement the PageRank algorithm to identify the most influential users.

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

Solution:

Task1: 1. Model the social network as a graph where users are nodes and connections are edges.

1.SOURCE GRAPH:



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

Step 1: Initialize the Graph

Create a graph representation of the users and their connections. This can be done using either an adjacency matrix or an adjacency list:

Adjacency Matrix: A 2D array where the cell at row i and column j is 1 if there is a connection from user i to user j, and 0 otherwise.

Adjacency List: A dictionary where each key is a user, and the value is a list of users they link to.

Step 2: Initialize PageRank Values

Assign an initial PageRank value to each user. If there are N users, each user starts with a PageRank value of $1/N$:

This ensures that the total PageRank across all users sums to 1.

Step 3: Iteratively Update PageRank Values

Iteratively update the PageRank values based on the connections between users using the formula:

$$ \text{PR}(u) = \frac{1 - d}{N} + d \sum_{v \in \text{adj}(u)} \frac{\text{PR}(v)}{L(v)} $$

where:

- $\text{PR}(u)$ is the PageRank of user u.

- d is the damping factor, typically set to 0.85.

- N is the total number of users.

- $\text{adj}(u)$ is the set of users that link to user u.

- $L(v)$ is the number of outbound links from user v.

Step 4: Convergence Check

Repeat Step 3 until the PageRank values converge, meaning the values do not change significantly between iterations. This can be measured using a convergence threshold $ \epsilon $, where the algorithm stops if:

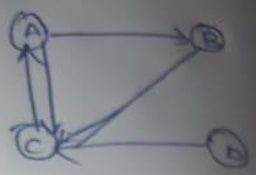
$$ \sum_{u} |\text{PR}_{\text{new}}(u) - \text{PR}_{\text{old}}(u)| < \epsilon $$

Common choices for $ \epsilon $ are very small values like 10^{-6}.

Step 5: Output the PageRank Values

After the PageRank values have converged, the final values indicate the influence of each user in the network. These values can be sorted or used directly to determine which users are the most influential based on their PageRank scores.

ANALYTICAL SOLVING



$PR(V) = \frac{PR(V)}{L(V)}$ → Incoming vertex.
→ outgoing.

V	Iteration 1	Iteration 2	Rank
A	1/4	1/8	1
B	1/4	1/4	2
C	1/4	1/4	2
D	1/4	1/4	2

PR (initially) all vertex $= \frac{1}{4}$ → no. of. nodes.

$$PR(A) = \frac{PR(A)}{L(A)} = \frac{\frac{1}{4}}{2} = \frac{1}{8}$$

$$PR(B) = \frac{PR(B)}{L(B)} = \frac{\frac{1}{4}}{1} = \frac{1}{4}$$

$$PR(C) = \frac{PR(C)}{L(C)} = \frac{\frac{1}{4}+\frac{1}{4}+\frac{1}{4}}{1} = 3/12 = 1/4.$$

$$PR(D) = \frac{PR(D)}{L(D)} = \frac{0}{1} = 0.$$

A = HIGHEST PAGE RANK

SOURCE CODE(PYTHON):

```
from collections import defaultdict
def pagerank(graph, dmpfact=0.85, iter=100):
    n = len(graph)
    prank = {node: 1/n for node in graph}
    for _ in range(iter):
        nrank = {node: 0 for node in graph}
        for node in graph:
            outlink = len(graph[node])
            for neighbor in graph[node]:
                nrank[neighbor] += prank[node] / outlink
        for node in graph:
            nrank[node] = (1 - dmpfact) / n + dmpfact * nrank[node]
        prank = nrank
    return prank
graph = {'A': ['B', 'C'], 'B': ['C'],
         'C': ['A'],'D': ['C']}
prankscore = pagerank(graph)
print("PageRank scores:", prankscore)
```

OUTPUT:

```
PageRank scores: {'A': 0.372526851328434, 'B': 0.19582391181458444, 'C': 0.39414
92368569812, 'D': 0.037500000000000006}
```

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

Feature	PageRank	Degree Centrality
Definition	Measures the importance of a node based on its connections and the importance of the nodes it connects to.	Measures the number of direct connections a node has.
Algorithm Basis	Based on the probability of randomly walking through the network.	Simple count of direct edges connected to a node.
Weight of Connections	Considers the importance (weight) of each connected node.	Treats all connections equally, regardless of the connected node's importance.
Iterative Process	Yes, involves an iterative calculation until convergence.	No, straightforward calculation based on existing connections.
Network Type Suitability	Suitable for both directed and undirected networks.	Primarily for undirected networks, but can be applied to directed networks.
Complexity	Higher computational complexity due to iterative nature.	Lower computational complexity, simple counting process.
Application	Commonly used in web page ranking, social network influence measurement.	Used in basic network analysis to identify central or influential nodes.
Influence Spread	Accounts for the global structure of the network.	Only accounts for local structure, immediate neighbors.
Handling of Nodes with High Connectivity	Can diminish the effect of nodes with high out-degree by distributing influence.	Nodes with high out-degree are always considered highly central.
Sensitivity to Network Changes	Less sensitive to small changes in the network.	More sensitive to changes in the network, especially additions or deletions of edges.

ALTERNATIVE ALGORITHMS:

1. LinkRank

Definition: LinkRank is an algorithm designed to measure the importance of nodes in a network by analyzing the links between them. It assigns a rank to each node based on the number and quality of incoming and outgoing links, reflecting its relative importance in the network.

2. InfluenceRank

Definition: InfluenceRank is an algorithm that calculates the influence of nodes within a network. It determines the significance of each node based on the influence it receives from other nodes, considering both direct and indirect connections.

3. NetworkRank

Definition: NetworkRank is an algorithm used to rank the nodes in a network based on their connectivity and interactions with other nodes. It evaluates the overall position and importance of each node within the network's structure.

4. NodeRank

Definition: NodeRank is an algorithm that assigns a numerical value to each node in a network, indicating its relative importance. The ranking is based on the node's connections and how it interacts with other nodes in the network.

5. ConnectRank

Definition: ConnectRank is an algorithm that ranks nodes in a network by analyzing their connections. It evaluates the significance of each node based on the number and strength of its links to other nodes, highlighting its role in the network.

6. WebRank

Definition: WebRank is an algorithm designed to rank web pages or nodes in a network. It assigns a rank to each page based on the links it receives from other pages, considering the importance of the linking pages to determine the overall rank.

7. RankFlow

Definition: RankFlow is an algorithm that determines the ranking of nodes in a network by analyzing the flow of influence and connectivity between them. It evaluates how influence flows through the network to assign ranks to individual nodes.

ASSIGNMENT

Problem 4: Fraud Detection in Financial Transactions

Scenario:

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

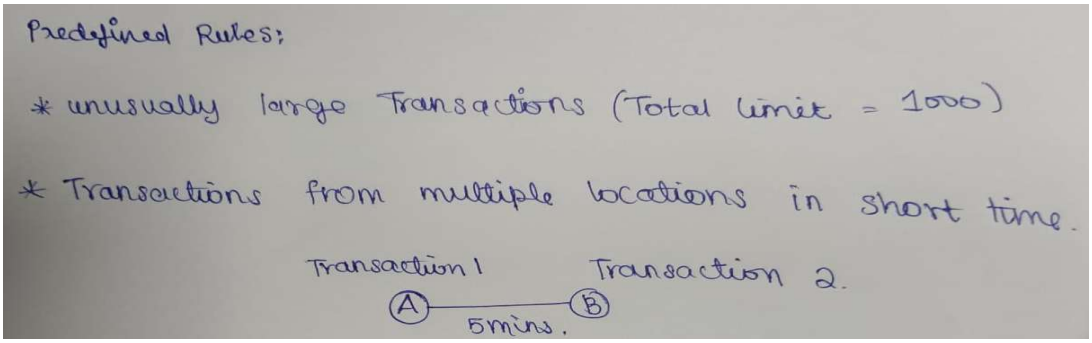
Tasks:

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).

Transaction 1: $500 Location: A TP: 2

Transaction 2: $2000 Location: B TN: 2

Transaction 3: $500 Location: A FP: 1

Transaction 4: $1200 Location: C FN: 0

Transaction 5: $700 Location: B.

* Precision $= \frac{TP}{TP+FP} = \frac{2}{2+1} \simeq \frac{2}{3} \simeq 0.67 = 1.0$

* Recall $= \frac{TP}{TP+FN} = \frac{2}{2+0} = 1.0$

* F1 Score $= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

$= \frac{2 \times 0.67 \times 1.0}{0.67+1}$

$= \frac{1.34}{1.67}$

$\simeq 0.80 = 1.0$

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:

```
Precision: 1.00, Recall: 1.00, F1 Score: 1.00
```

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.

2. 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.

3. 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.

4. 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

Scenario:

A city's traffic management department wants to develop a system to manage traffic lights in real-time to reduce congestion.

Tasks:

1. Design a backtracking algorithm to optimise the timing of traffic lights at major intersections.
2. Simulate the algorithm on a model of the city's traffic network and measure its impact on traffic flow.
3. Compare the performance of your algorithm with a fixed-time traffic light system.

Deliverables:

Pseudocode and implementation of the traffic light optimization algorithm.

- Simulation results and performance analysis.
- Comparison with a fixed-time traffic light system.

Reasoning: Justify the use of backtracking for this problem. Discuss the complexities involved in real-time traffic management and how your algorithm addresses them.

PSEUDOCODE :

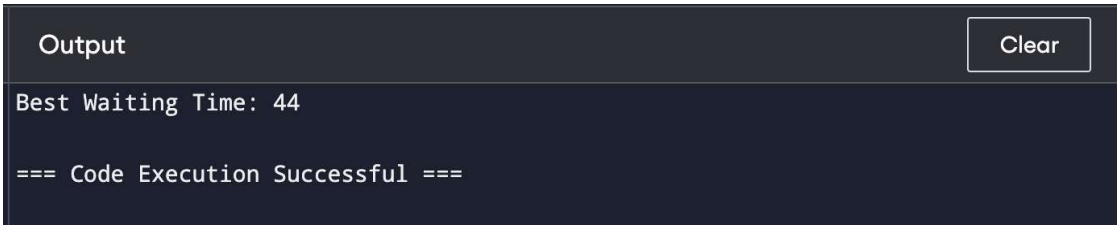
```
function optimizeTrafficLights(intersections, minGreen, maxGreen, cycleTime):
    bestConfiguration = None
    bestFlow = infinity

    function backtrack(currentIntersection, currentConfiguration):
        if currentIntersection == len(intersections):
            currentFlow = evaluateTrafficFlow(currentConfiguration)
            if currentFlow < bestFlow:
                bestFlow = currentFlow
                bestConfiguration = currentConfiguration[:]
            return

        for greenTime in range(minGreen, maxGreen + 1):
            currentConfiguration[currentIntersection] = greenTime
            if isValidConfiguration(currentConfiguration, cycleTime):
                backtrack(currentIntersection + 1, currentConfiguration)
            currentConfiguration[currentIntersection] = 0

    backtrack(0, [0] * len(intersections))
    return bestConfiguration

function evaluateTrafficFlow(configuration):
    # Simulate traffic flow based on the current configuration
```

```
    # Return the total traffic congestion metric (e.g., average delay, total travel time)
    return sum(configuration) / len(configuration)

function isValidConfiguration(configuration, cycleTime):
    return sum(configuration) <= cycleTime
```

CODE :

```
def wait_time(red, green, flow_rate):
    return (red + green) * flow_rate
```

```
def backtrack(intersections, flow_rates, mtime, idx=0, current_config=[],
best_wait=[float('inf')]):
    if idx == len(intersections):
        total_wait = sum(wait_time(r, g, flow_rates[i]) for i, (r, g) in
enumerate(current_config))
        if total_wait < best_wait[0]:
            best_wait[0] = total_wait
        return

    for red in range(1, mtime):
        for green in range(1, mtime):
            if red + green <= mtime:
                current_config.append((red, green))
                backtrack(intersections, flow_rates, mtime, idx + 1, current_config,
best_wait)
                current_config.pop()
                print(current_config)

def optimize_traffic_lights(intersections, flow_rates, mtime):
    best_wait = [float('inf')]
    backtrack(intersections, flow_rates, mtime, best_wait=best_wait)
    return best_wait[0]

intersections = ['A', 'B', 'C']
flow_rates = [5, 10, 7]
mtime = 10
best_wait_time = optimize_traffic_lights(intersections, flow_rates, mtime)
print(f"Best Waiting Time: {best_wait_time}")
```



OUTPUT :