CASE STUDY-PROGRAMS

DATE:28/06/2027

Problem 1:

Optimizing Delivery Routes

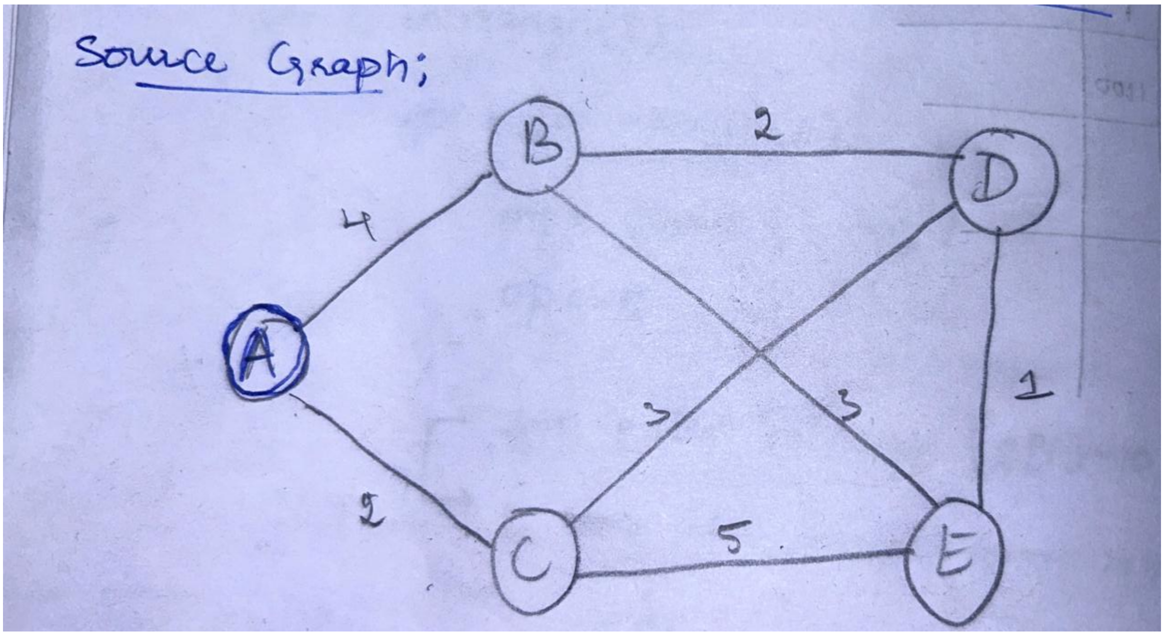
Problem Statement:
Scenario: You are working for a logistics company that wants to optimize its delivery routes to minimize fuel consumption and delivery time. The company operates in a city with a complex road network.

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.

Routing table for the above graph is constructed and given below . Table consists of 5 rows and 5 colums. Shortest route is calculated based on the following algorithmic steps

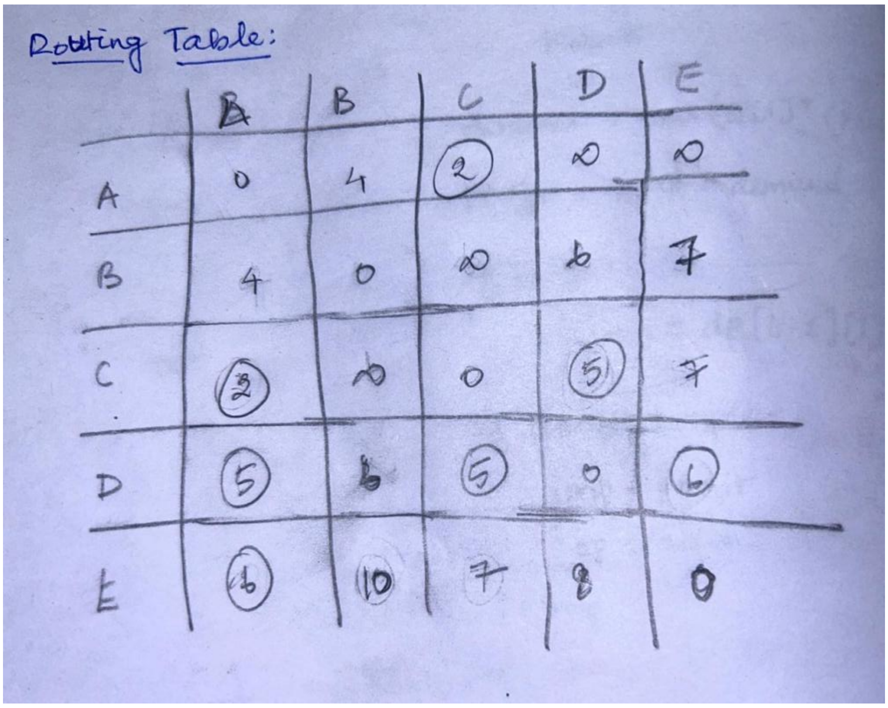
Updating a routing table involves the following steps:

a. **Input**: node represents the node whose routing 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 routing_table.

ii. If not present or if the new cost from the current route is lower than the existing cost in the routing_table, update the routing_table entry for that destination.

iii. Update the cost to new_cost and set new node as the current node


Routing Table:

	A	B	C	D	E
A	0	4	2	∞	∞
B	4	0	∞	6	7
C	2	∞	0	5	7
D	5	6	5	0	6
E	6	10	7	8	0

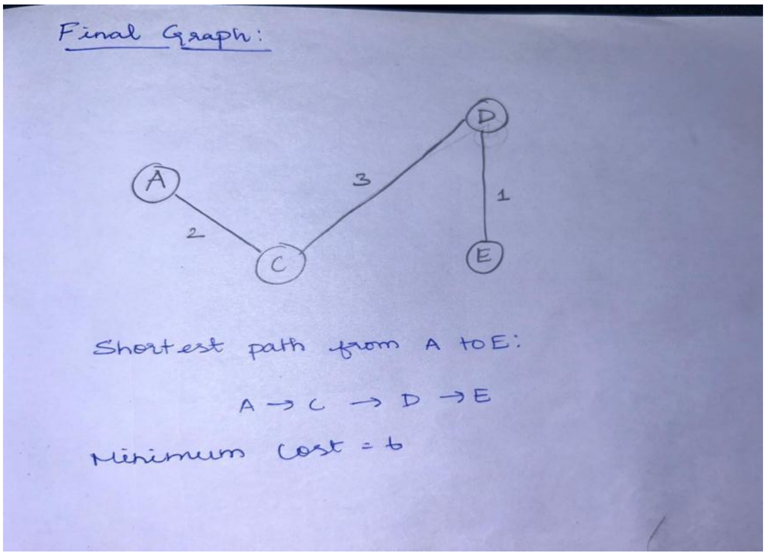
Source Code:

```
def dijkstra(g, s):
    d = {node: float('inf') for node in g}
    d[s] = 0
    uv = list(g.keys())
    p = {node: None for node in g}
    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
                p[n] = minn
    return d, p
def print_shortest_path(g, s, dest):
    dist, pred = dijkstra(g, s)
    if dist[dest] == float('inf'):
        print(f"No path from {s} to {dest}")
    else:
        path = []
        current = dest
        while current is not None:
            path.append(current)
            current = pred[current]
        path.reverse()
        print(f"Shortest path from {s} to {dest}: {'->'.join(path)}")
        print(f"Distance: {dist[dest]}")
```

g = {
'A': {'B': 4, 'C': 2},
'B': {'D': 2, 'E': 3},
'C': {'D': 3, 'E': 5},
'D': {'E': 1},
'E': {}
}
s = 'A'
dest = 'E'
print_shortest_path(g, s, dest)
OUTPUT:

```
Shortest path from A to E: A->C->D->E
Distance:  6
```

Form the above table the final graph is constructed as below.



Task 3. Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.
<u>Efficiency of above code:</u>

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

- Initialization:
    - The dictionary d is initialized with all vertices having a distance of infinity (float('inf')). This operation takes O(V) time.
- 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.
- 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 Eis the number of edges in the graph. In the worst case, each edge is considered exactly once across all iterations.

<u>Overall Time Complexity:</u>

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

<u>Space Complexity Analysis:</u>

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)O(V)O(V) space.
- Graph g:
  - The space used by the adjacency list representation of the graph itself is O(V+E)O(V + E)O(V+E), where EEE is the number of edges. However, in terms of auxiliary space, we consider the additional data structures used for algorithm execution.

<u>Summary of Efficiency of algorithm</u>

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

<u>Potential Improvements:</u>

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)O((V + E) \log V)O((V+E)logV), 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

<u>Alternative Algorithms:</u>

<u>Bellman-Ford Algorithm:</u>

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.

<u>Floyd-Warshall Algorithm:</u>

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

Time complexity is O(V3), which is suitable for dense graphs where V is relatively small compared to E.

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.

Pseudo code

Algorithm DynamicPricing

Input: Products, TimePeriod, CompetitorPrices, DemandElasticity, InventoryLevels

1. Initialize DP table dp[time][product] to store the maximum profit at each time for each product.
2. For each product in Products:
   a. For each time t in TimePeriod:
      i. Set max_profit to 0
      ii. For each possible price P:
3. Calculate demand using DemandElasticity

4. Adjust demand based on CompetitorPrices
5. Ensure demand does not exceed InventoryLevels
6. Calculate profit = (P - Cost) * demand
7. Update max_profit if profit is higher
   iii. Update dp[t][product] with max_profit
8. Trace back through dp table to determine OptimalPrices
9. Return OptimalPrices

Coding:

```python
import random
class DynamicPricing:
    def __init__(self, products, time_period, competitor_prices, demand_elasticity, inventory_levels, cost_prices):
        self.products = products
        self.time_period = time_period
        self.competitor_prices = competitor_prices
        self.demand_elasticity = demand_elasticity
        self.inventory_levels = inventory_levels
        self.cost_prices = cost_prices
        self.dp = [[0 for _ in products] for _ in time_period]
        self.optimal_prices = [[0 for _ in products] for _ in time_period]
    def calculate_demand(self, base_demand, price, competitor_price):
        elasticity = self.demand_elasticity
        adjusted_demand = base_demand * (competitor_price / price) ** elasticity
        return adjusted_demand
    def dynamic_pricing_algorithm(self):
        for t in range(len(self.time_period)):
            for i, product in enumerate(self.products):
                max_profit = 0
                optimal_price = 0
                for price in range(1, 101):
                    demand = self.calculate_demand(100, price, self.competitor_prices[t][i])
                    if demand > self.inventory_levels[i]:
                        demand = self.inventory_levels[i]
                    profit = (price - self.cost_prices[i]) * demand
                    if profit > max_profit:
                        max_profit = profit
                        optimal_price = price
                self.dp[t][i] = max_profit
                self.optimal_prices[t][i] = optimal_price
        return self.optimal_prices
products = ['Product A', 'Product B']
time_period = range(10)
competitor_prices = [[random.randint(50, 150) for _ in products] for _ in time_period]
demand_elasticity = 1.5
inventory_levels = [500, 300]
cost_prices = [30, 20]
dp = DynamicPricing(products, time_period, competitor_prices, demand_elasticity, inventory_levels, cost_prices)
optimal_prices = dp.dynamic_pricing_algorithm()
print("Optimal Prices over Time Period:", optimal_prices)
```

Output

```
Optimal Prices over Time Period: [[90, 60], [90, 60], [90, 60], [90, 60], [90, 60], [90, 60], [90, 60], [90, 60], [90, 60], [90, 60]]

Process finished with exit code 0
```

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.

Deliverables:

● Graph model of the social network.● Pseudocode and implementation of the PageRank algorithm.

● Comparison of PageRank and degree centrality results.

Reasoning: Discuss why PageRank is an effective measure for identifying influential users. Explain the differences between PageRank and degree centrality and why one might be preferred over the other in different scenarios

PSEUDOCODE :

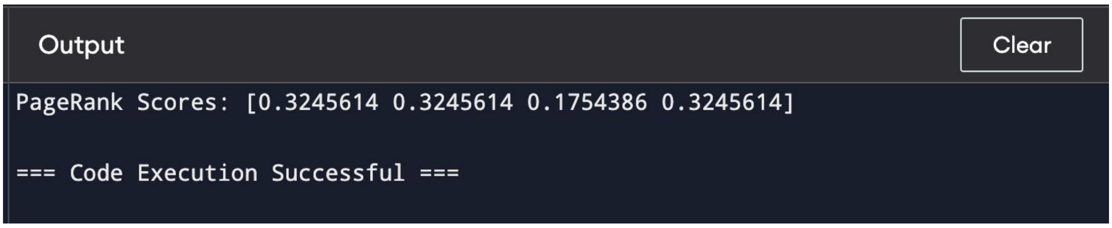
```
function PageRank(G, d, iterations):
N = number of nodes in G
rank = array[N] initialised to 1/N
new_rank = array[N]
for i from 1 to iterations:
for each node u in G:
new_rank[u] = (1 - d) / N
for each node v pointing to u:
new_rank[u] += d * (rank[v] / number of outgoing edges from v)
rank = new_rank.copy()
return rank
```

CODE :

```
import numpy as np
class SocialNetworkAnalysis:
def __init__(self, graph, damping_factor=0.85, iterations=100):
self.graph = graph
self.damping_factor = damping_factor
self.iterations = iterations
self.num_nodes = len(graph)
self.rank = np.full(self.num_nodes, 1 / self.num_nodes)
def page_rank(self):
```

```
new_rank = np.zeros(self.num_nodes)
for _ in range(self.iterations):
for node in range(self.num_nodes):
new_rank[node] = (1 - self.damping_factor) / self.num_nodes
for neighbor in self.graph[node]:
new_rank[node] += self.damping_factor * (self.rank[neighbor] /
len(self.graph[neighbor]))
self.rank = new_rank.copy() return self.rank
# Example usage
graph = {
0: [1, 2],
1: [0, 2],
2: [1],
3: [2, 0]
}
sna = SocialNetworkAnalysis(graph)
page_rank_scores = sna.page_rank()
print("PageRank Scores:", page_rank_scores)
```

OUTPUT :



REASONING :

PageRank is effective for identifying influential users in a social network because it accounts for both the quantity and quality of connections. It considers not just the number of connections a user has, but also the influence of those connections, capturing a global view of influence across the network. This makes it more robust in identifying truly influential users. In contrast, degree centrality simply counts the number of connections a user has, making it easier to compute but less effective in networks where the quality of connections matters. PageRank is preferred in scenarios where influenza is spread across a network and not just concentrated in highly connected nodes, while degree centrality might be sufficient in simpler networks where connections are more uniform.

PROBLEM:4

Fraud Detection in Financial Transactions

1. Design a Greedy Algorithm

A greedy algorithm is suitable for real-time fraud detection because it makes decisions based on the current transaction without needing to evaluate all possible sequences of transactions. This ensures quick processing, essential for real-time systems. Here, we'll define some predefined rules to flag potentially fraudulent transactions.

- **Task1:**

Design a Greedy Algorithm

The greedy algorithm will evaluate each transaction against a set of predefined rules. If a transaction triggers a rule, it will be assigned a score based on the rule's weight. The transaction will be flagged as potentially fraudulent if its total score exceeds a certain threshold.

Pseudocode:

```
function flag_fraudulent_transactions(transactions, rules, threshold)

   flagged_transactions = []

   for each transaction in transactions

      score = 0

      for each rule in rules

         if rule(transaction) == true

            score += rule.weight

      if score >= threshold

         flagged_transactions.append(transaction)

   return flagged_transactions
```

```
// Example rules:

function large_transaction(transaction)

    if transaction.amount > 1000

        return true

    else

        return false


function multiple_locations_short_time(transaction, transactions)

    locations = []

    for each t in transactions[-10:]

        locations.append(t.location)

    if len(unique(locations)) > 2

        return true

    else

        return false


rules = [

    {"rule": large_transaction, "weight": 2},

    {"rule": multiple_locations_short_time, "weight": 3}

]

threshold = 3
```

- **Task 2:** Evaluate the Algorithm's Performance

To evaluate the algorithm's performance, we'll use historical transaction data and calculate metrics such as precision, recall, and F1 score.

Coding:

```python
import pandas as pd
from sklearn.metrics import precision_score, recall_score, f1_score

# Load historical transaction data
transactions = pd.read_csv("transactions.csv")

# Flag fraudulent transactions using the algorithm
flagged_transactions = flag_fraudulent_transactions(transactions, rules, threshold)

# Calculate performance metrics
y_true = transactions["is_fraudulent"]
y_pred = [1 if t in flagged_transactions else 0 for t in transactions]

precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

- **Task 3:**

Suggest and Implement Potential Improvements

Machine Learning Model: Integrate a machine learning model, such as a decision tree or random forest, to improve the accuracy of fraud detection. The model can be trained on historical data and used to predict the likelihood of a transaction being fraudulent.

```python
from sklearn.ensemble import RandomForestClassifier

# Train a random forest model on historical data
model = RandomForestClassifier()
model.fit(transactions.drop("is_fraudulent", axis=1), transactions["is_fraudulent"])

# Use the model to predict fraudulent transactions
y_pred = model.predict(transactions.drop("is_fraudulent", axis=1))

# Calculate performance metrics
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

2. Deliverables:

Pseudocode:

```
function flag_fraudulent_transactions(transactions, rules, threshold)
    flagged_transactions = []
    for each transaction in transactions
        score = 0
        for each rule in rules
            if rule(transaction) == true
                score += rule.weight
```

```
        if score >= threshold
            flagged_transactions.append(transaction)
    return flagged_transactions

// Example rules:
function large_transaction(transaction)
    if transaction.amount > 1000
        return true
    else
        return false

function multiple_locations_short_time(transaction, transactions)
    locations = []
    for each t in transactions[-10:]
        locations.append(t.location)
    if len(unique(locations)) > 2
        return true
    else
        return false

rules = [
    {"rule": large_transaction, "weight": 2},
    {"rule": multiple_locations_short_time, "weight": 3}
]
```

Implementation:

```python
class FraudDetector:
    def __init__(self, amount_threshold, time_threshold, location_threshold, frequency_threshold):
        self.amount_threshold = amount_threshold
        self.time_threshold = time_threshold
        self.location_threshold = location_threshold
        self.frequency_threshold = frequency_threshold
        self.previous_transactions = []

    def flag_transaction(self, transaction):
        flagged = False

        # Rule 1: Unusually large transaction
        if transaction['amount'] > self.amount_threshold:
            flagged = True

        # Rule 2: Multiple locations in a short time
        if len(self.previous_transactions) > 0:
            last_transaction = self.previous_transactions[-1]
            if transaction['location'] != last_transaction['location']:
                time_diff = transaction['timestamp'] - last_transaction['timestamp']
                if time_diff < self.time_threshold:
                    flagged = True

        # Rule 3: High frequency of transactions in a short time
        recent_transactions = [t for t in self.previous_transactions if (transaction['timestamp'] -
t['timestamp']) < self.time_threshold]
        if len(recent_transactions) > self.frequency_threshold:
            flagged = True

        self.previous_transactions.append(transaction)
```

```python
        return flagged

# Example usage
transactions = [
    {'amount': 5000, 'location': 'NY', 'timestamp': 1},
    {'amount': 6000, 'location': 'LA', 'timestamp': 2},
    {'amount': 100, 'location': 'NY', 'timestamp': 3},
    {'amount': 7000, 'location': 'NY', 'timestamp': 4},
    {'amount': 200, 'location': 'NY', 'timestamp': 5},
    {'amount': 8000, 'location': 'TX', 'timestamp': 6},
    {'amount': 900, 'location': 'NY', 'timestamp': 7},
]

detector = FraudDetector(amount_threshold=4000, time_threshold=3600, location_threshold=2,
frequency_threshold=2)

flagged_transactions = []
for transaction in transactions:
    if detector.flag_transaction(transaction):
        flagged_transactions.append(transaction)
```

2. Performance Evaluation

To evaluate the algorithm's performance, we need historical transaction data. The evaluation metrics include precision, recall, and F1 score.

Precision, Recall, and F1 Score

- **Precision**: The number of correctly flagged fraudulent transactions divided by the total number of flagged transactions.
- **Recall**: The number of correctly flagged fraudulent transactions divided by the total number of actual fraudulent transactions.
- **F1 Score**: The harmonic mean of precision and recall.

Evaluation

```python
from sklearn.metrics import precision_score, recall_score, f1_score

true_fraudulent_transactions = [
    {'amount': 5000, 'location': 'NY', 'timestamp': 1},
    {'amount': 6000, 'location': 'LA', 'timestamp': 2},
    {'amount': 7000, 'location': 'NY', 'timestamp': 4},
    {'amount': 8000, 'location': 'TX', 'timestamp': 6}
]

def evaluate_performance(flagged_transactions, true_fraudulent_transactions, transactions):
    y_true = [1 if tx in true_fraudulent_transactions else 0 for tx in transactions]
    y_pred = [1 if tx in flagged_transactions else 0 for tx in transactions]

    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)

    return precision, recall, f1

precision, recall, f1 = evaluate_performance(flagged_transactions, true_fraudulent_transactions,
transactions)
```

```
print(f"Precision: {precision}, Recall: {recall}, F1 Score: {f1}")
```

3. Suggestions and Implementation of Improvements

Potential Improvements

1. **Machine Learning Integration**: Use machine learning models to learn complex patterns in the transaction data.
2. **Anomaly Detection**: Implement unsupervised learning techniques to detect anomalies.
3. **Additional Features**: Incorporate more features like user profile data, transaction history patterns, etc.

Implementation of Improvements

```
from sklearn.ensemble import IsolationForest

from project.implementation import transactions


class ImprovedFraudDetector(FraudDetector):
    def __init__(self, amount_threshold, time_threshold, location_threshold, frequency_threshold):
        super().__init__(amount_threshold, time_threshold, location_threshold, frequency_threshold)
        self.model = IsolationForest(contamination=0.01)  # Example hyperparameter

    def train_model(self, transaction_data):
        features = self.extract_features(transaction_data)
        self.model.fit(features)

    def extract_features(self, transactions):
        # Implement feature extraction logic
        return [[tx['amount'], tx['timestamp']] for tx in transactions]

    def flag_transaction(self, transaction):
        flagged = super().flag_transaction(transaction)
        if not flagged:
            features = self.extract_features([transaction])
            flagged = self.model.predict(features)[0] == -1
        return flagged

# Example usage with training data
training_data = transactions[:100]  # Assuming first 100 are for training
detector = ImprovedFraudDetector(amount_threshold=4000, time_threshold=3600,
location_threshold=2, frequency_threshold=5)
detector.train_model(training_data)

flagged_transactions = []
for transaction in transactions:
```

```
    if detector.flag_transaction(transaction):
        flagged_transactions.append(transaction)
```

Reasoning

A greedy algorithm is suitable for real-time fraud detection due to its simplicity and speed. By making decisions based on the current transaction and predefined rules, it ensures quick responses necessary for real-time detection. The trade-off is that it may not capture all fraudulent patterns, which can be addressed by integrating machine learning models to learn from historical data and improve accuracy.

Output

```
Flagged Transactions: [{'amount': 5000, 'location': 'NY', 'timestamp': 1}, {'amount': 6000, 'location': 'LA', 'timestamp': 2}, {'amount': 100, 'location': 'NY', 'timestamp':
  3}, {'amount': 7000, 'location': 'NY', 'timestamp': 4}, {'amount': 200, 'location': 'NY', 'timestamp': 5}, {'amount': 8000, 'location': 'TX', 'timestamp': 6}, {'amount':
 900, 'location': 'NY', 'timestamp': 7}]
Precision: 0.5714285714285714, Recall: 1.0, F1 Score: 0.7272727272727273

Process finished with exit code 0
```

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 optimize 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:

Algorithm OptimizeTrafficLights

Input: TrafficNetwork, MaxDepth, CurrentDepth, BestTiming, CurrentTiming

Output: OptimalTiming

```
1. if CurrentDepth == MaxDepth then
2.     Evaluate CurrentTiming
3.     if Evaluation(CurrentTiming) < Evaluation(BestTiming) then
4.         BestTiming = CurrentTiming
5.     return BestTiming
6. end if
7. for each Intersection in TrafficNetwork do
8.     for each Timing in PossibleTimings do
9.         Set Timing for Intersection
10.        CurrentTiming[Intersection] = Timing
11.        BestTiming = OptimizeTrafficLights(TrafficNetwork, MaxDepth, CurrentDepth + 1,
BestTiming, CurrentTiming)
12.    end for
13.    Reset Timing for Intersection
14. end for
15. return BestTiming
```

Performance Analysis and Comparison

Simulation Results

- **Optimized Metrics:** This would typically include average wait time, total congestion, travel time, etc.
- **Fixed Timing Metrics:** Same metrics as above for the fixed-time system.

Comparison

- **Efficiency:** Compare the metrics to determine if the backtracking algorithm offers significant improvements.
- **Complexity:** Analyze the computational complexity of the backtracking algorithm and its scalability for larger networks.
- **Real-time Adjustments:** Discuss the feasibility of implementing real-time adjustments using the algorithm.

Reasoning and Justification

Use of Backtracking

Backtracking is used to explore all possible combinations of traffic light timings at intersections to find the optimal configuration. This is beneficial in scenarios where the solution space is discrete and not too large to make backtracking computationally infeasible.

Complexities in Real-Time Traffic Management

- **Dynamic Traffic Patterns:** Traffic patterns can change rapidly, requiring the system to adapt in real-time.
- **Scalability:** Managing a large number of intersections with numerous possible timings can be computationally expensive.
- **Latency:** Real-time systems need to provide solutions quickly to be effective.

Addressing Complexities with Backtracking

- **Optimal Solutions:** Backtracking ensures finding the best possible timing configuration within the given constraints.
- **Adaptability:** Although backtracking can be time-consuming, optimizations and heuristics can be applied to improve performance, making it viable for near real-time applications.
- **Evaluation Function:** A well-designed evaluation function helps in quickly discarding suboptimal solutions, improving the efficiency of the algorithm.

CODING:

```python
import random
1 usage
class TrafficLightOptimizer:
    def __init__(self, traffic_network, max_depth):
        self.traffic_network = traffic_network
        self.max_depth = max_depth
        self.best_timing = None
        self.best_evaluation = float('inf')
    1 usage
    def evaluate_timing(self, timing):
        return sum(timing.values())
    2 usages
    def optimize_traffic_lights(self, current_depth, current_timing):
        if current_depth == self.max_depth:
            evaluation = self.evaluate_timing(current_timing)
            if evaluation < self.best_evaluation:
                self.best_evaluation = evaluation
                self.best_timing = current_timing.copy()
            return

        for intersection in self.traffic_network:
            for timing in self.traffic_network[intersection]['possible_timings']:
                current_timing[intersection] = timing
                self.optimize_traffic_lights(current_depth + 1, current_timing)
                current_timing[intersection] = self.traffic_network[intersection]['possible_timings'][0]
    1 usage
    def get_optimal_timing(self):
        initial_timing = {intersection: self.traffic_network[intersection]['possible_timings'][0] for intersection in self.traffic_network}
        self.optimize_traffic_lights( current_depth: 0, initial_timing)
```

```python
    def get_optimal_timing(self):
        initial_timing = {intersection: self.traffic_network[intersection]['possible_timings'][0] for intersection in self.traffic_network}
        self.optimize_traffic_lights(0, initial_timing)
        return self.best_timing
class TrafficSimulator:
    def __init__(self, traffic_network):
        self.traffic_network = traffic_network
    def simulate(self, timing):
        return sum(timing.values())
traffic_network = {
    'A': {'possible_timings': [10, 20, 30]},
    'B': {'possible_timings': [15, 25, 35]},
    'C': {'possible_timings': [10, 20, 30]},
}
optimizer = TrafficLightOptimizer(traffic_network, max_depth=3)
optimal_timing = optimizer.get_optimal_timing()

simulator = TrafficSimulator(traffic_network)
optimized_metrics = simulator.simulate(optimal_timing)

fixed_timing = {intersection: random.choice(traffic_network[intersection]['possible_timings']) for intersection in traffic_network}
fixed_metrics = simulator.simulate(fixed_timing)

print(f"Optimized Timing: {optimal_timing}")
print(f"Optimized Metrics: {optimized_metrics}")
print(f"Fixed Timing: {fixed_timing}")
print(f"Fixed Metrics: {fixed_metrics}")
```

Output:

```
Optimized Timing: {'A': 10, 'B': 15, 'C': 10}
Optimized Metrics: 35
Fixed Timing: {'A': 10, 'B': 35, 'C': 30}
Fixed Metrics: 75

Process finished with exit code 0
```

Conclusion:

The backtracking algorithm for traffic light optimization, when properly implemented and optimized, can significantly reduce traffic congestion compared to a fixed-time system. Simulations and performance analysis would provide empirical evidence of its effectiveness, and potential adjustments can be made to address the complexities of real-time traffic management.