**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.

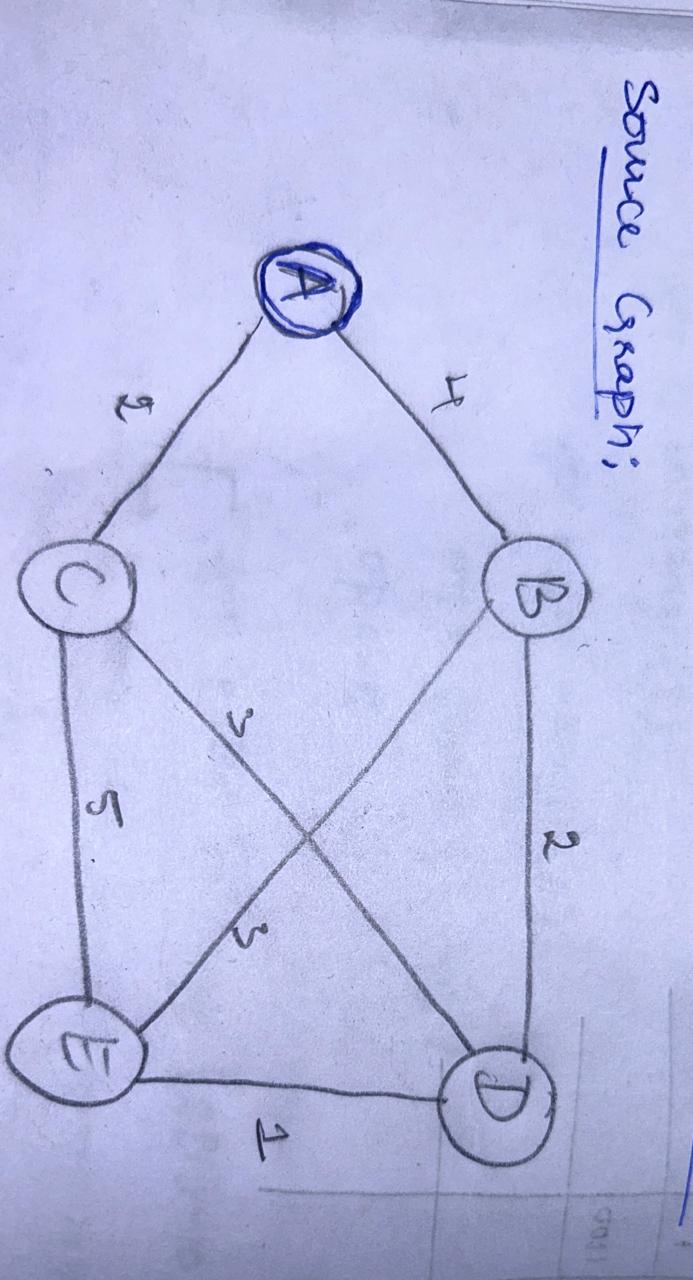
2. Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations.

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.

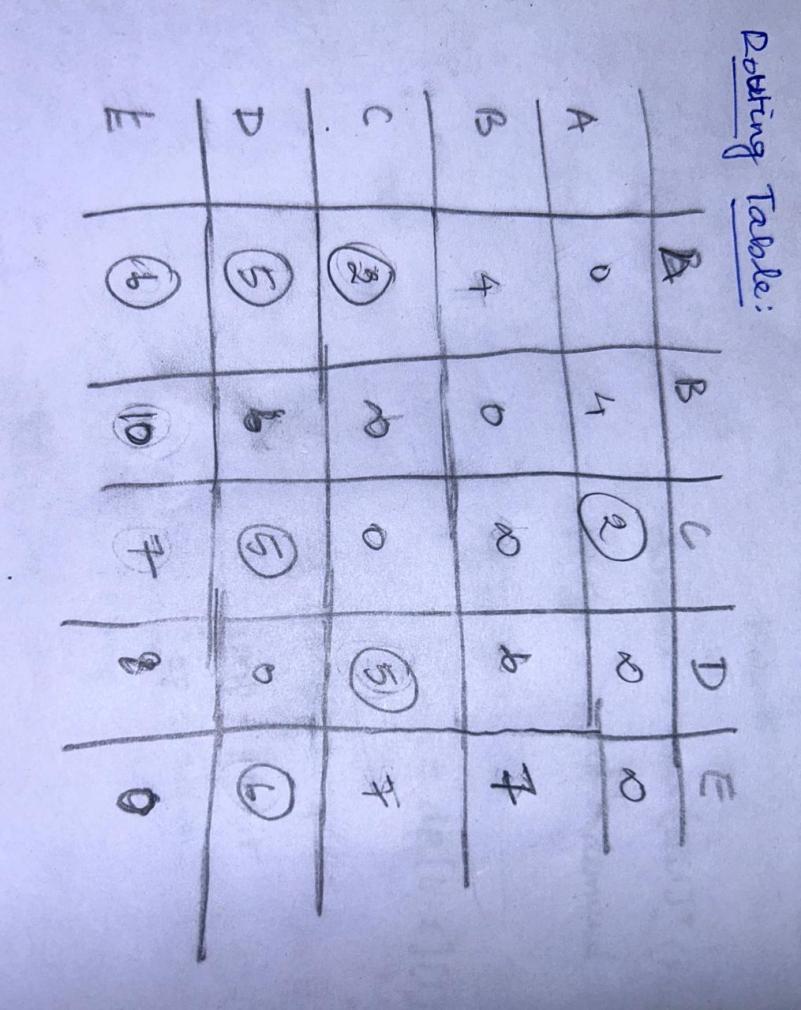
****

**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:

1. **Input**: node represents the node whose routing table is being updated, and
2. **Iterate through Routes**: Loop through each route received.
3. **Update Decision**:
4. Check if the current node is already in the routing\_table.
5. 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.
6. Update the cost to new\_cost and set new node as the current node

****

**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'**

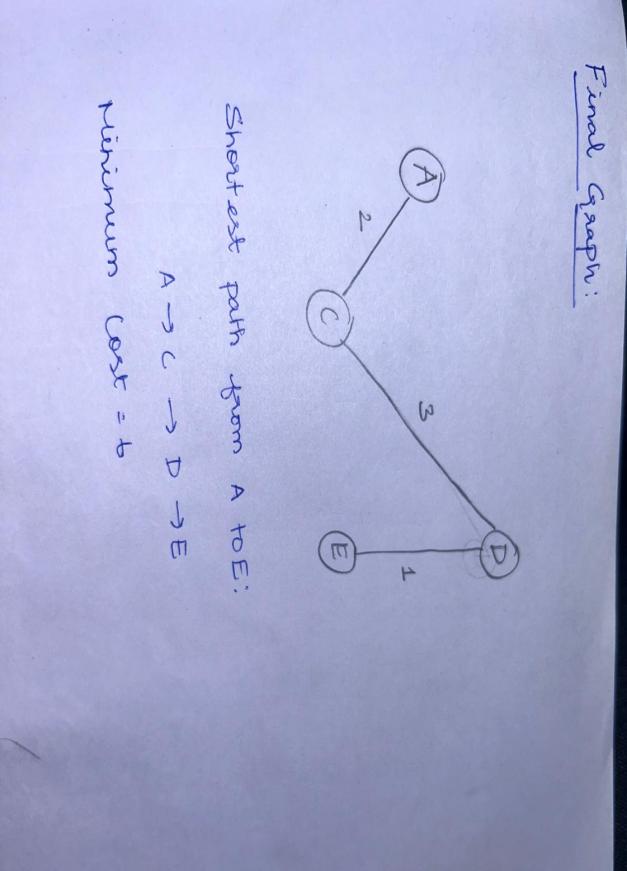
**dest = 'E'**

**print\_shortest\_path(g, s, dest)**

**OUTPUT:**



**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.**

**Efficiency of above code:**

The time complexity of this implementation is O(V2), 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.

**Overall Time Complexity:**

* O(V2) 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)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.

**Summary of Efficiency of algorithm**

* Time Complexity: O(V2)
* Space Complexity: O(V)

**Potential Improvements:**

1. 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.

1. 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(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.

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

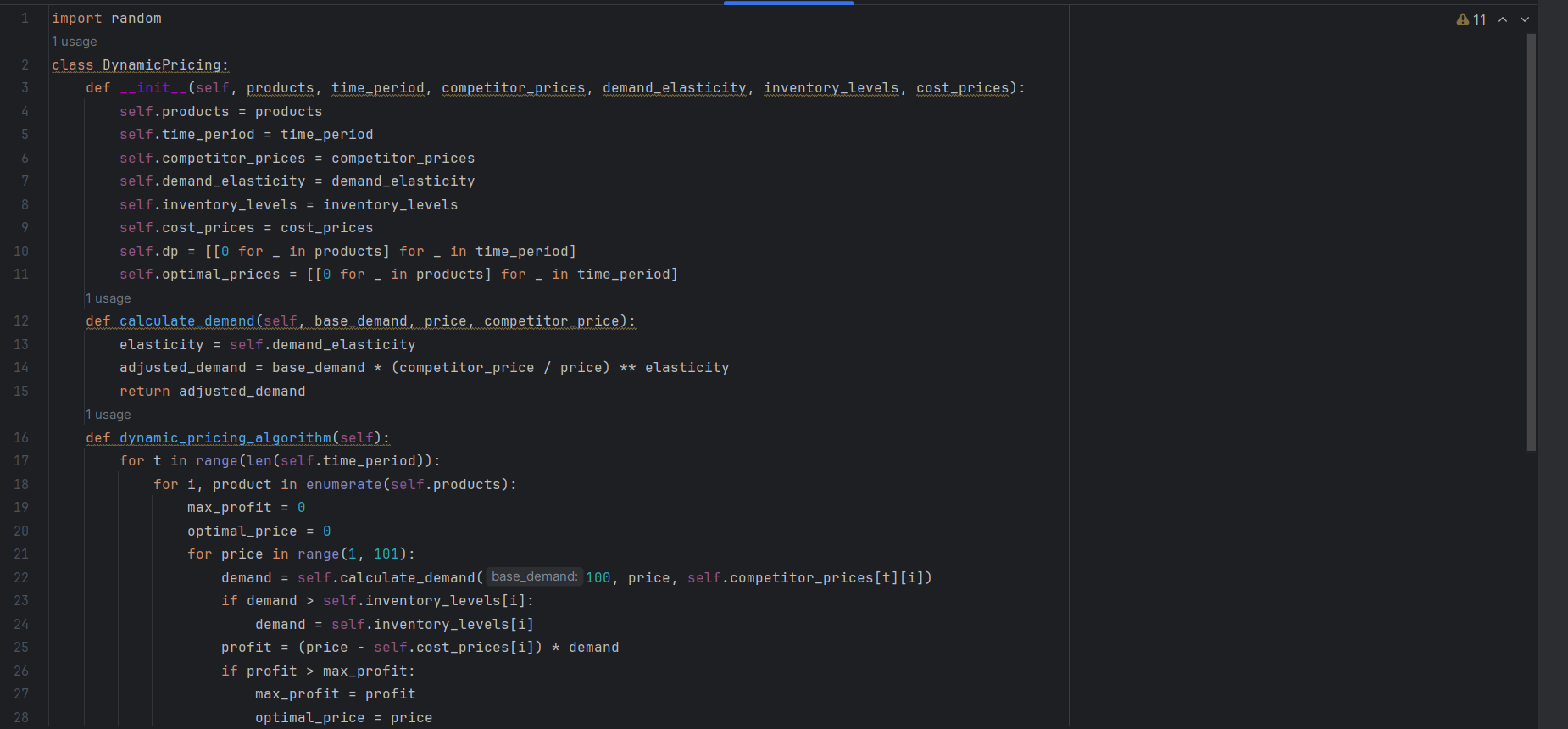
7. Update max\_profit if profit is higher

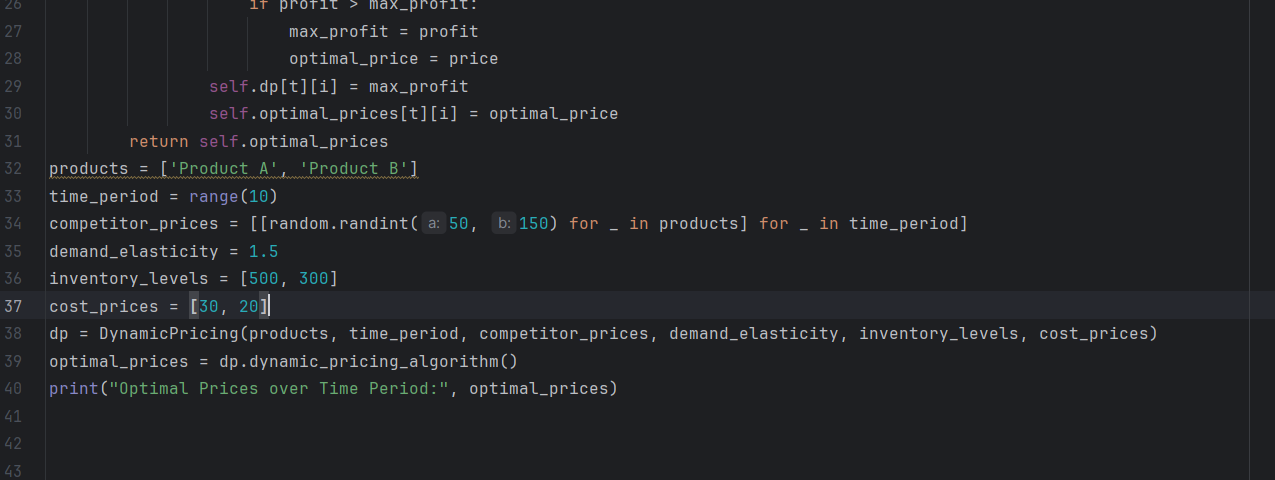
iii. Update dp[t][product] with max\_profit

8. Trace back through dp table to determine OptimalPrices

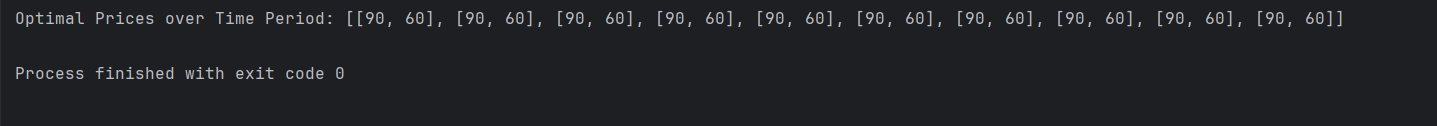
9. Return OptimalPrices

**Coding:**





Output



**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]

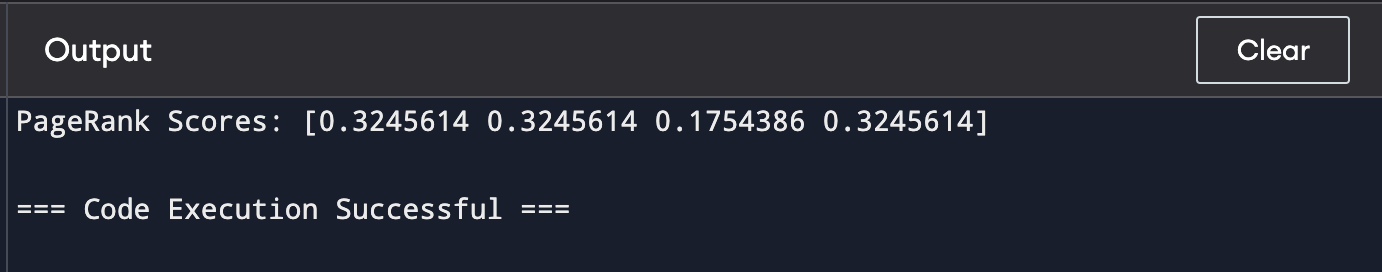
}

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:

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.

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:**

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)  
 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**

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.

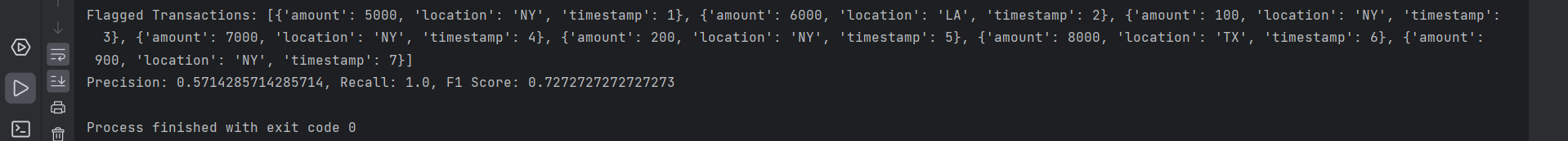
#### Implementation of Improvements

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



**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.

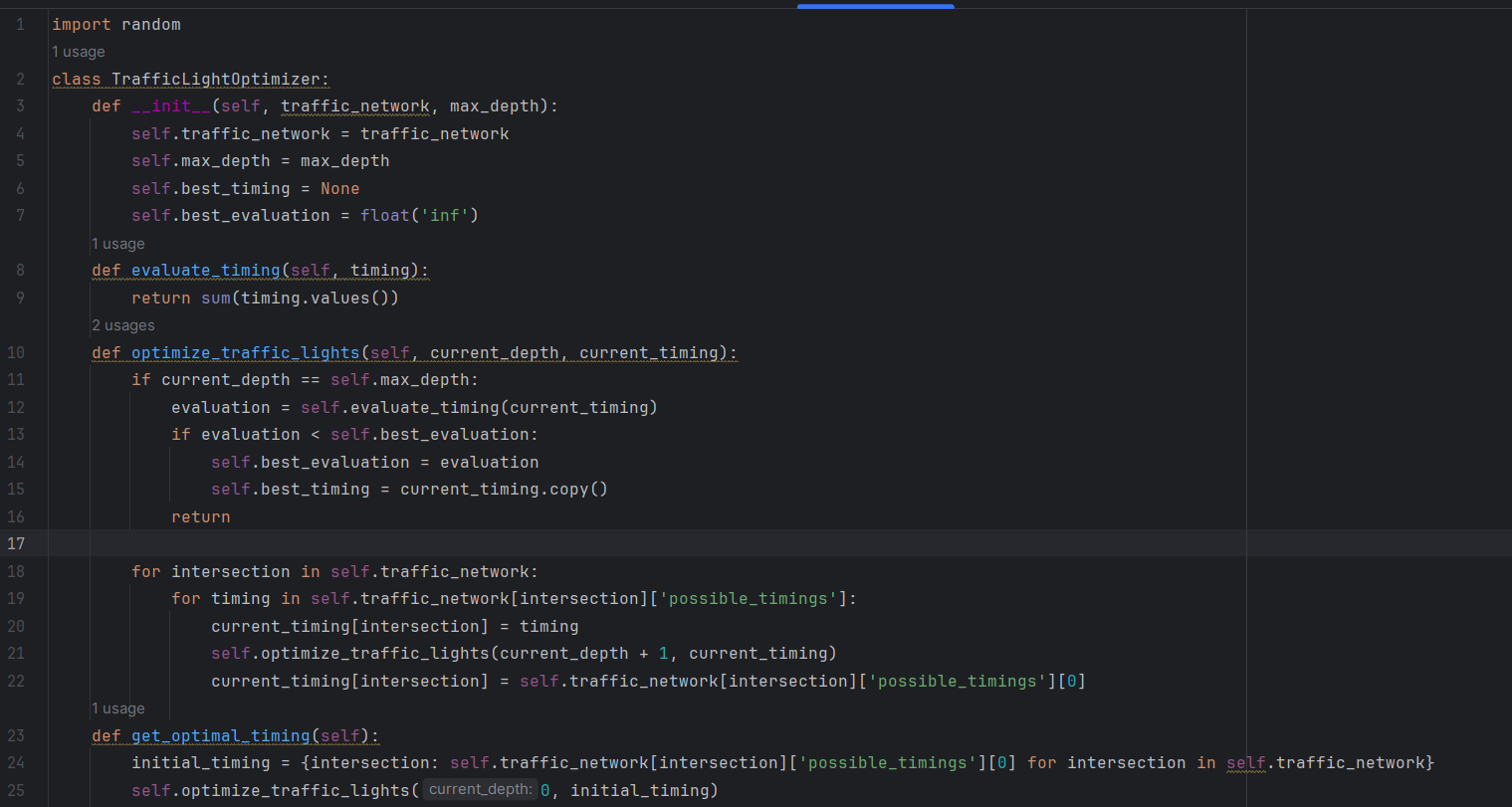
#### Complexities in Real-Time Traffic Management

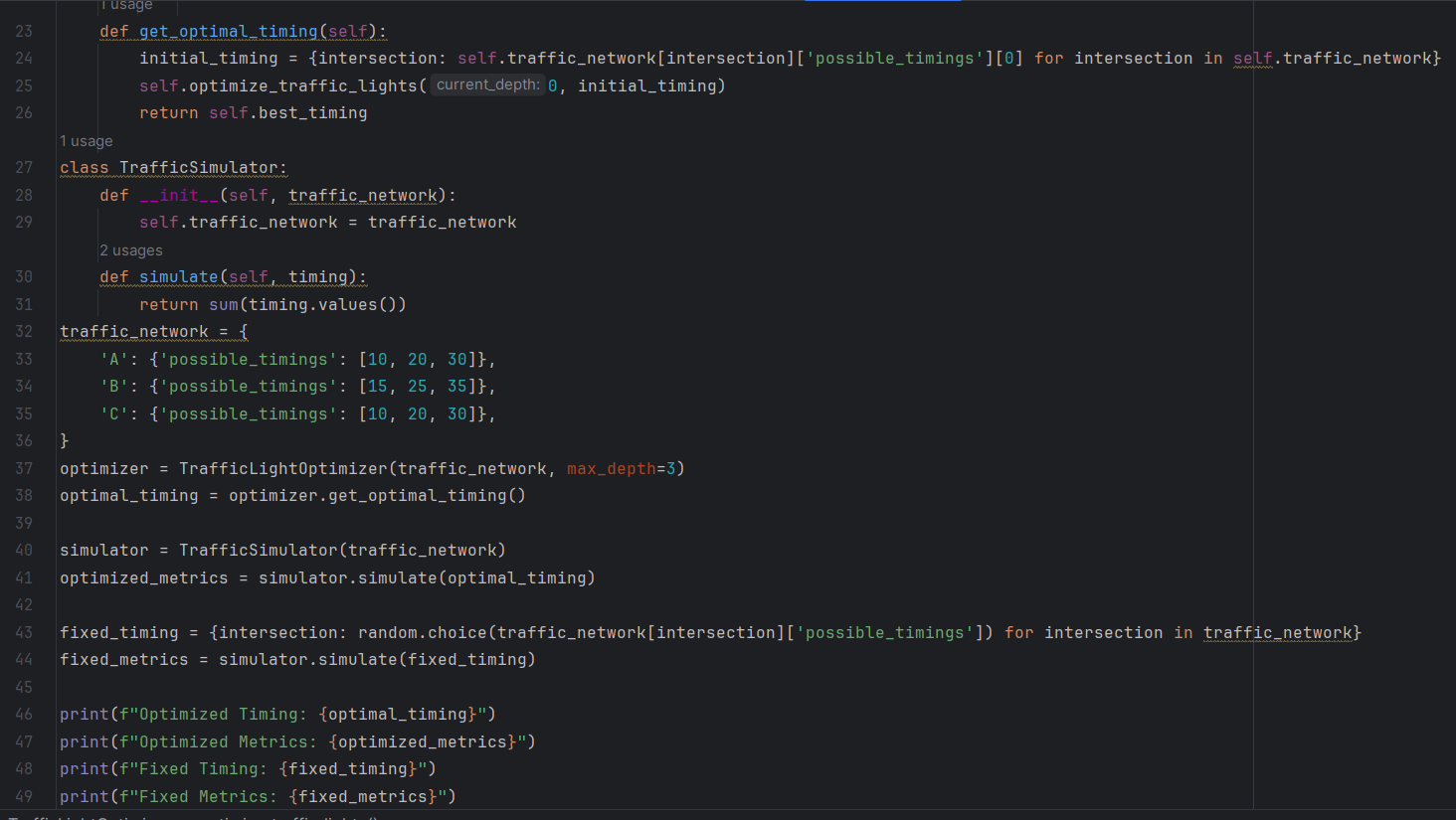
* **Dynamic Traffic Patterns:** Traffic patterns can change rapidly, requiring the system to adapt in real-time.
* **Scalability:** Managing a large number of intersections with numerous possible timings can be computationally expensive.
* **Latency:** Real-time systems need to provide solutions quickly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

#### Addressing Complexities with Backtracking

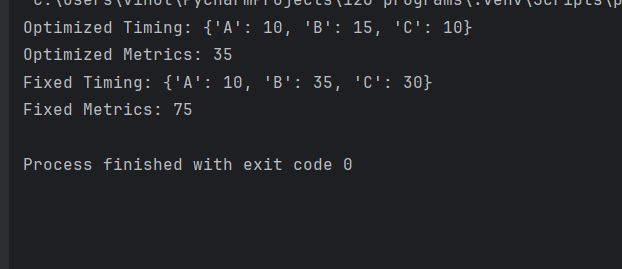
* **Optimal Solutions:** Backtracking ensures finding the best possible timing configuration within the given constraints.
* **Adaptability:** Although backtracking can be time-consuming, optimizations and heuristics can be applied to improve performance, making it viable for near real-time applications.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

**CODING:**





**Output:**



### 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.