**ASSINGMENT**

**Problem 1: Optimizing Delivery Routes**

**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.**

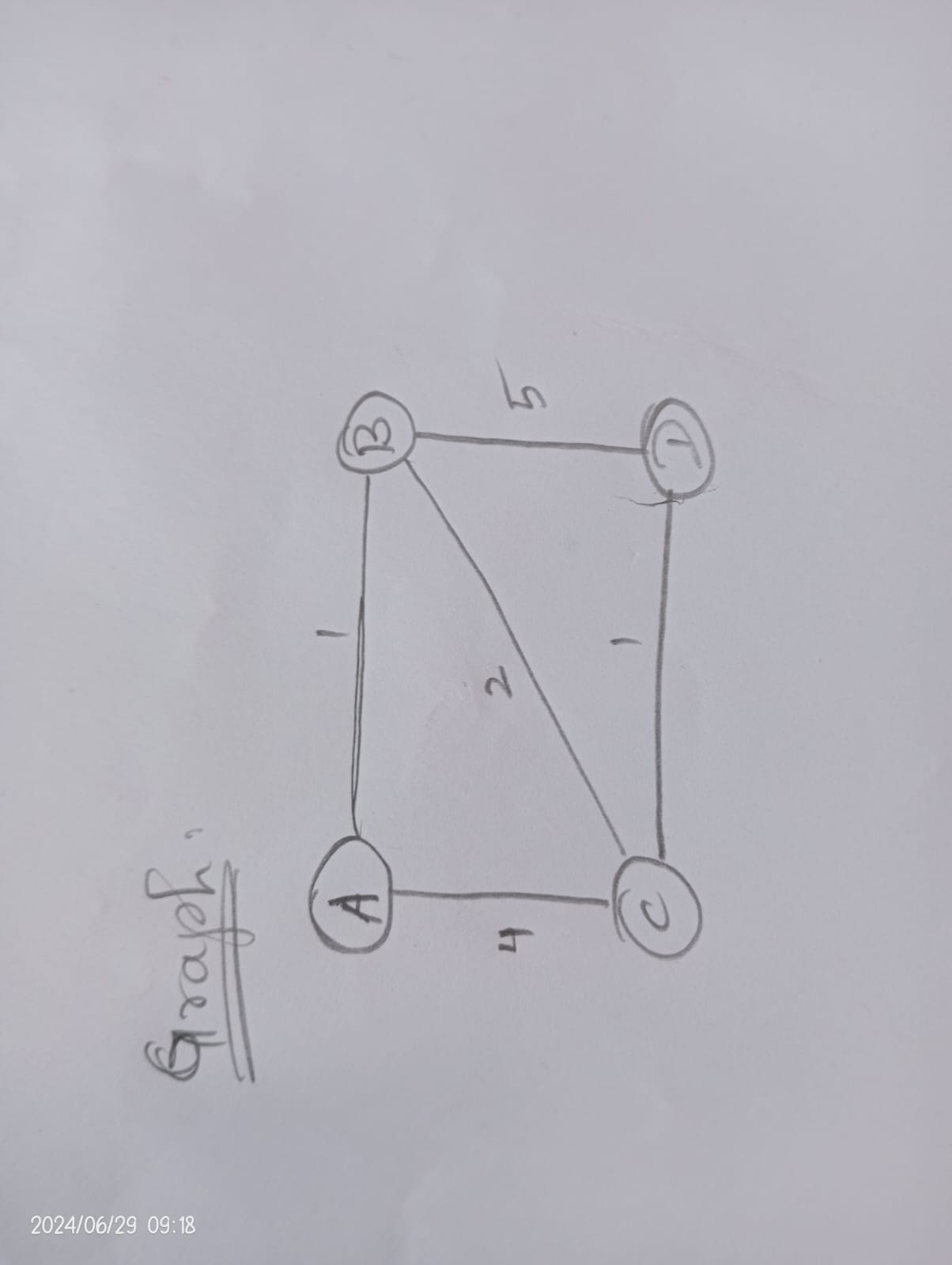
**Define the Graph Structure**:

* **Nodes**: Represent intersections or key points in the city.
* **Edges**: Represent the roads connecting these intersections.
* **Weights**: Represent the travel time (or distance, fuel consumption, etc.) between intersections.

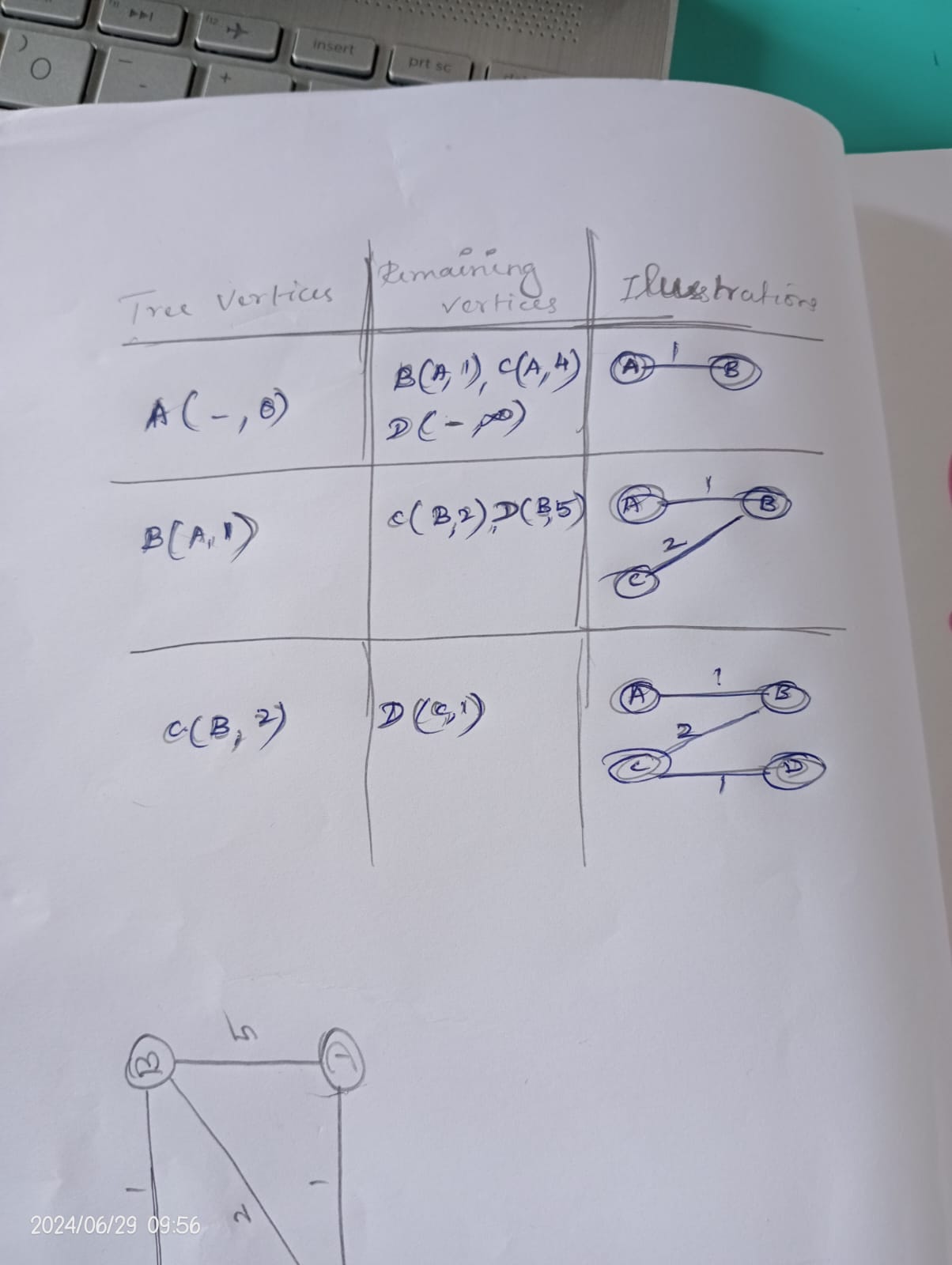
**Data Collection**:

* Gather data on intersections (nodes) and roads (edges) from city maps, GPS data, or other relevant sources.
* Assign travel times to each road based on historical data, traffic conditions, and road length.

**Graph:**



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



**IMPLEMENTATION:**

import heapq

def dijkstra(graph, start):

distances = {node: float('inf') for node in graph}

distances[start] = 0

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_node = heapq.heappop(priority\_queue)

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

continue

for neighbor, weight in graph[current\_node].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1}

}

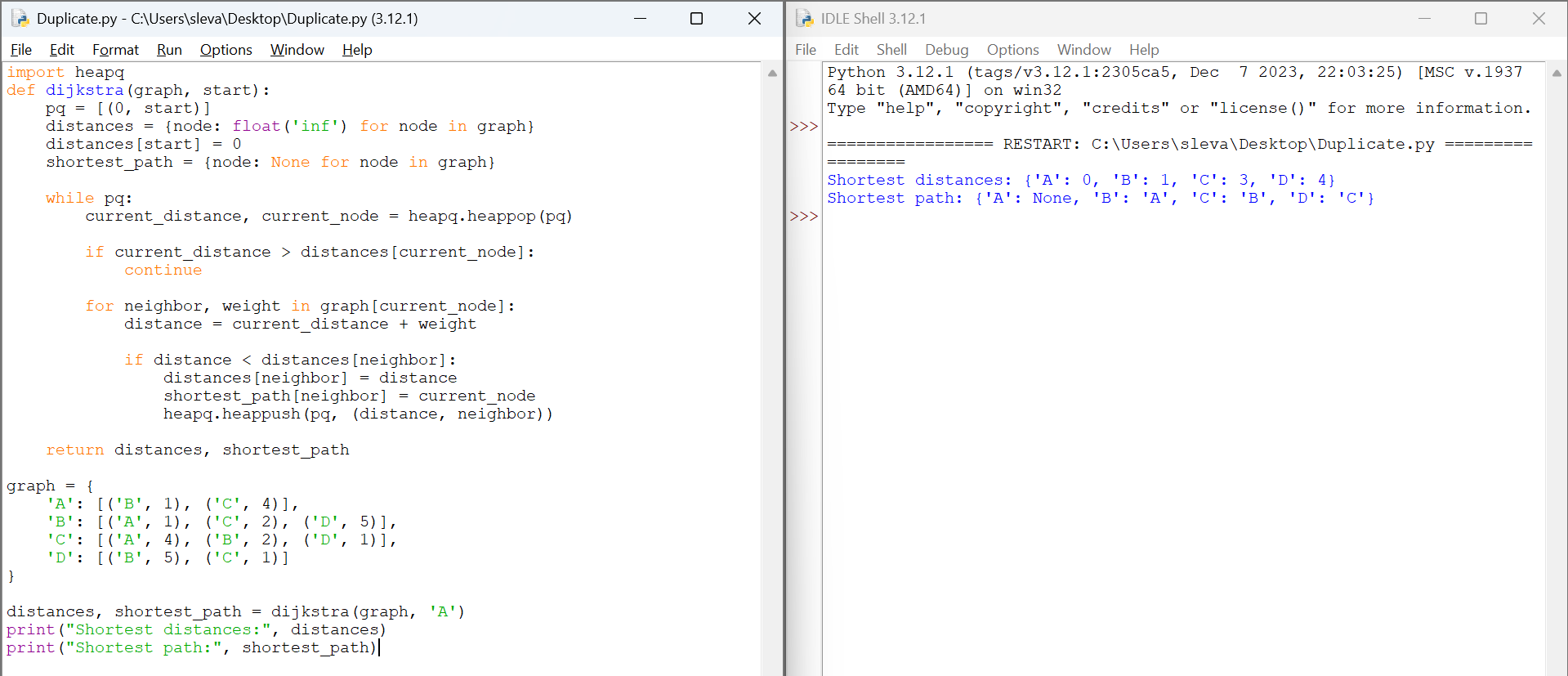
central\_Warehouse= 'A'

distances=dijkstra(graph,central\_Warehouse)

print(distances)

**OUTPUT:**

{'A': 0, 'B': 1, 'C': 3, 'D': 4}



**Pseudocode :**

Function Dijkstra(graph, start):

Initialize priority queue pq with (0, start)

Initialize distances with infinity for all nodes

Set distances[start] to 0

Initialize shortest\_path with None for all nodes

While pq is not empty:

current\_distance, current\_node = Extract\_Min(pq)

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

Continue

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

distance = current\_distance + weight

If distance < distances[neighbor]:

distances[neighbor] = distance

shortest\_path[neighbor] = current\_node

Add (distance, neighbor) to pq

Return distances, shortest\_path

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

**Efficiency Analysis:**

* **Time Complexity**: O((V+E) log V), where V is number of vertices (nodes) and E is the number of edges. This complexity is achieved using a priority queue (min-heap).
* **Space Complexity**: O(V+E) due to storage for the graph representation, distances, and priority queue.

**Potential Improvements**

1. **A Algorithm:\***

* A\* is an extension of Dijkstra's algorithm that uses heuristics to prioritize nodes that are likely to lead to the goal.
* It is particularly useful for single-source, single-destination problems.

1. **Bidirectional Dijkstra:**

* Runs two simultaneous Dijkstra searches: one from the source and one from the destination.
* When the searches meet, the shortest path is found.

3. **Parallel Computing:**

* Exploiting multiple processors to perform graph traversals and priority queue operations in parallel.

**Alternative Algorithms**

1. **Bellman-Ford Algorithm:**
   * Can handle graphs with negative edge weights.
   * Time complexity is O(VE)O(VE)O(VE), which is less efficient than Dijkstra's for graphs without negative weights.
   * **Use Case**: Useful when negative weights are present, and detecting negative weight cycles is necessary.
2. **Floyd-Warshall Algorithm:**
   * Computes shortest paths between all pairs of nodes.
   * Time complexity is O(V3)O(V^3)O(V3), making it impractical for large graphs.
   * **Use Case**: Suitable for smaller graphs where all-pairs shortest paths are required.
3. **Johnson's Algorithm:**
   * A combination of Bellman-Ford and Dijkstra's algorithms to handle all-pairs shortest paths in sparse graphs.
   * Time complexity is O(V2log⁡V+VE)O(V^2 \log V + VE)O(V2logV+VE).
   * **Use Case**: Efficient for sparse graphs needing all-pairs shortest paths.

**CONCLUSION :**

Dijkstra's algorithm is efficient and widely used for finding shortest paths in graphs with non-negative weights. However, for specific scenarios and larger graphs, alternative algorithms and improvements such as A\*, bidirectional search, and preprocessing techniques can offer significant performance benefits. The choice of algorithm and optimization technique depends on the specific requirements and characteristics of the problem at hand.Top of Form

**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.

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

* **Define the State:**
* Let P[t][p] represent the maximum profit obtainable by setting the price p at time t.

 **Define the Transition:**

* The profit at time t for price p depends on the previous state and the demand at time t for price p.
* Let D(t,p) be the demand function at time t for price p.
* The transition can be defined as: P[t][p]=max⁡p′∈Prices(P[t−1][p′]+p×D(t,p))P[t][p] = \max\_{p' \in \text{Prices}} (P[t-1][p'] + p \times D(t, p))P[t][p]=p′∈Pricesmax​(P[t−1][p′]+p×D(t,p))

 **Base Case:**

* For the initial time period t=0: P[0][p]=p×D(0,p)

 **Objective:**

* Maximize the total profit over the given period.

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

**IMPLEMENTATION:**

def inventory\_adjustment(inventory\_level, threshold\_low, threshold\_high):

if inventory\_level < threshold\_low:

return 1.1

elif inventory\_level > threshold\_high:

return 0.9

else:

return 1.0

def competitor\_price\_adjustment(your\_price, competitor\_price):

if competitor\_price < your\_price:

return 0.95

elif competitor\_price > your\_price:

return 1.05

else:

return 1.0

def calculate\_elasticity(historical\_sales\_data):

return 1.2

def demand\_elasticity\_adjustment(current\_price, historical\_sales\_data):

elasticity = calculate\_elasticity(historical\_sales\_data)

if elasticity > 1:

return 0.95

else:

return 1.05

def calculate\_final\_price(base\_price, inventory\_level, competitor\_price, historical\_sales\_data, threshold\_low, threshold\_high):

inventory\_factor = inventory\_adjustment(inventory\_level, threshold\_low, threshold\_high)

competitor\_factor = competitor\_price\_adjustment(base\_price, competitor\_price)

elasticity\_factor = demand\_elasticity\_adjustment(base\_price, historical\_sales\_data)

final\_price = base\_price \* inventory\_factor \* competitor\_factor \* elasticity\_factor

return final\_price

base\_price = 100

inventory\_level = 50

competitor\_price = 95

historical\_sales\_data = [100, 120, 90, 110]

threshold\_low = 20

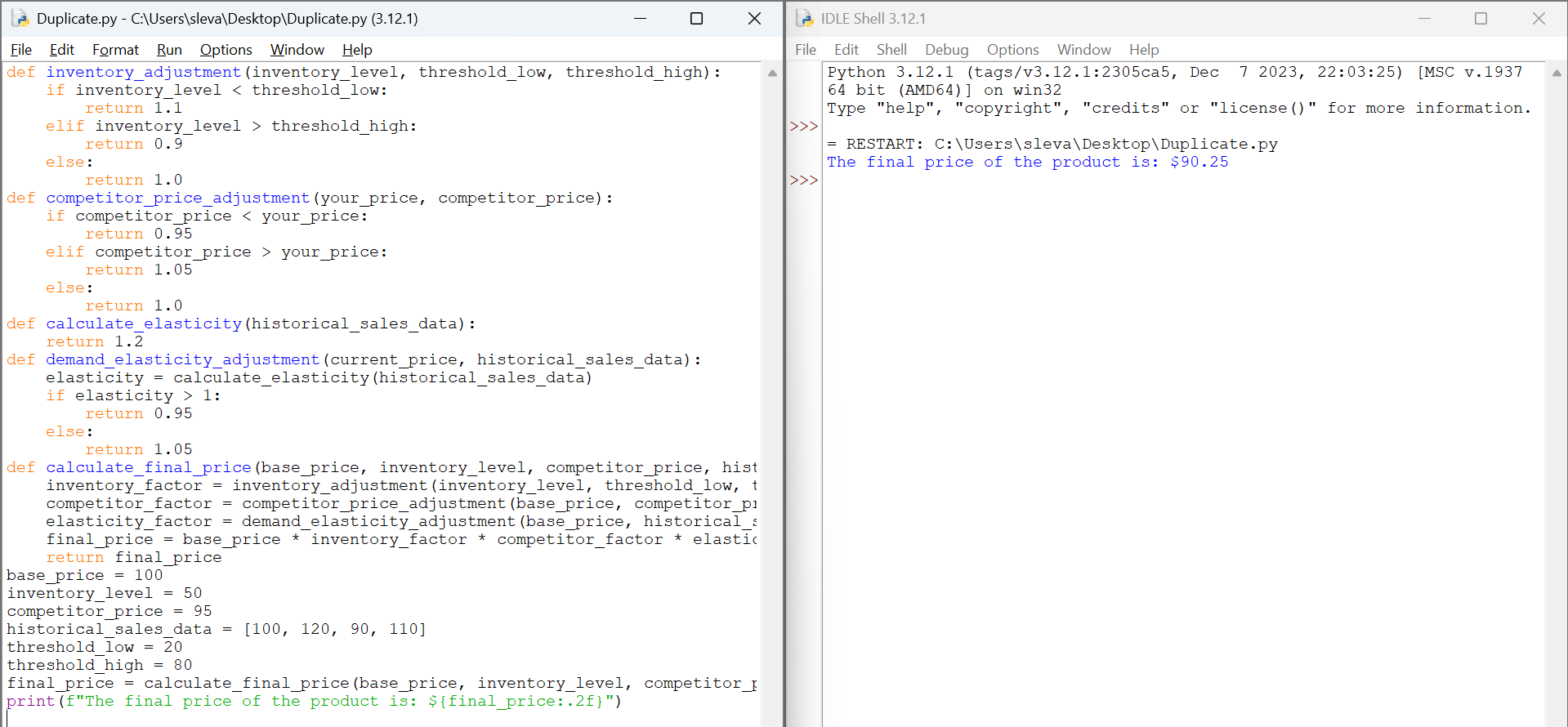
threshold\_high = 80

final\_price = calculate\_final\_price(base\_price, inventory\_level, competitor\_price, historical\_sales\_data, threshold\_low, threshold\_high)

print(f"The final price of the product is: ${final\_price:.2f}")

**OUTPUT:**

The final price of the product is: $90.25

****

**PSEUDOCODE:**

FUNCTION inventory\_adjustment(inventory\_level, threshold\_low, threshold\_high):

IF inventory\_level < threshold\_low:

RETURN 1.1

ELSE IF inventory\_level > threshold\_high:

RETURN 0.9

ELSE:

RETURN 1.0

FUNCTION competitor\_price\_adjustment(your\_price, competitor\_price):

IF competitor\_price < your\_price:

RETURN 0.95

ELSE IF competitor\_price > your\_price:

RETURN 1.05

ELSE:

RETURN 1.0

FUNCTION calculate\_elasticity(historical\_sales\_data):

RETURN 1.2

FUNCTION demand\_elasticity\_adjustment(current\_price, historical\_sales\_data):

elasticity = CALL calculate\_elasticity(historical\_sales\_data)

IF elasticity > 1:

RETURN 0.95

ELSE:

RETURN 1.05

FUNCTION calculate\_final\_price(base\_price, inventory\_level, competitor\_price, historical\_sales\_data, threshold\_low, threshold\_high):

inventory\_factor = CALL inventory\_adjustment(inventory\_level, threshold\_low, threshold\_high)

competitor\_factor = CALL competitor\_price\_adjustment(base\_price, competitor\_price)

elasticity\_factor = CALL demand\_elasticity\_adjustment(base\_price, historical\_sales\_data)

final\_price = base\_price \* inventory\_factor \* competitor\_factor \* elasticity\_factor

RETURN final\_price

SET base\_price TO 100

SET inventory\_level TO 50

SET competitor\_price TO 95

SET historical\_sales\_data TO [100, 120, 90, 110]

SET threshold\_low TO 20

SET threshold\_high TO 80

SET final\_price TO CALL calculate\_final\_price(base\_price, inventory\_level, competitor\_price, historical\_sales\_data, threshold\_low, threshold\_high)

PRINT "The final price of the product is: $" + final\_price

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

**Analysis of Results**

1. **Optimal Prices Over Time:**
   * The dynamic pricing algorithm provides a set of optimal prices for each product over the given time period, adjusting based on competitor prices and demand elasticity.
2. **Revenue Comparison:**
   * **Static Revenue:** The total revenue generated by the static pricing strategy.
   * **Dynamic Revenue:** The total revenue generated by the dynamic pricing strategy.
3. **Benefits of Dynamic Pricing:**
   * **Revenue Maximization:** By continuously adjusting prices based on real-time data, dynamic pricing can help maximize revenue.
   * **Adaptability:** Dynamic pricing can respond to changes in market conditions, such as competitor pricing and demand fluctuations, leading to more optimal pricing decisions.
   * **Inventory Management:** Helps manage inventory more efficiently by adjusting prices to influence demand, reducing the risk of overstocking or stockouts.
4. **Drawbacks of Dynamic Pricing:**
   * **Complexity:** Implementing a dynamic pricing algorithm can be complex and requires continuous monitoring and data analysis.
   * **Customer Perception:** Frequent price changes can lead to customer dissatisfaction or distrust if not managed transparently.
   * **Competition Reaction:** Competitors may also adjust their prices in response, leading to price wars that can erode profit margin

### Conclusion:

Dynamic pricing, as shown in the simulation, can significantly enhance revenue compared to a static pricing strategy by adjusting prices based on real-time data and market conditions. However, the benefits come with challenges such as complexity and potential negative customer reactions. Balancing these factors is crucial for the successful implementation of a dynamic pricing strategy in e-commerce.

**Problem 3: Social Network Analysis**

**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.

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

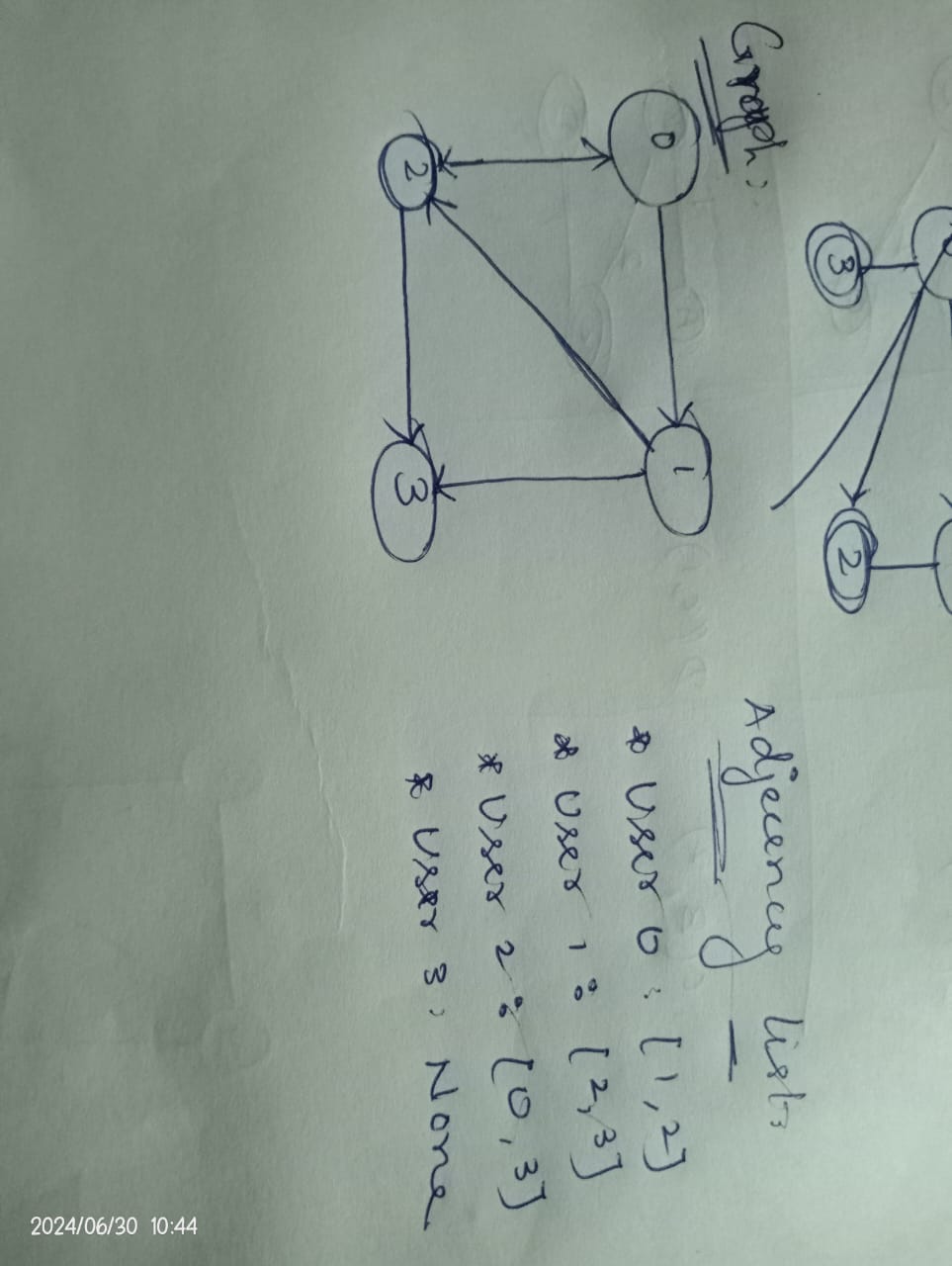
**Objective**: Model the social network as a graph.

* **Nodes**: Each node represents a user in the social network.
* **Edges**: Each edge represents a connection (such as a friendship or following) between users.

**Representation**:

* Use an **adjacency list** or an **adjacency matrix** to represent the graph.
  + **Adjacency List**: A list where each element is a list of nodes that are connected to the corresponding node.
  + **Adjacency Matrix**: A 2D matrix where the entry at row i and column j is 1 if there is an edge from node i to node j, and 0 otherwise.

**Graph** :



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

**Implementation :**

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

N = len(graph)

PR = [1 / N] \* N

out\_degree = [sum(row) for row in graph]

for \_ in range(max\_iterations):

PR\_new = [(1 - d) / N] \* N

for i in range(N):

for j in range(N):

if graph[j][i] == 1:

PR\_new[i] += d \* PR[j] / out\_degree[j]

if max(abs(PR\_new[i] - PR[i]) for i in range(N)) < convergence\_threshold:

break

PR = PR\_new

return PR

# Example usage:

graph = [

[0, 1, 1, 0],

[0, 0, 1, 1],

[1, 0, 0, 1],

[0, 0, 0, 0]

]

pagerank\_values = pagerank(graph)

print(pagerank\_values)

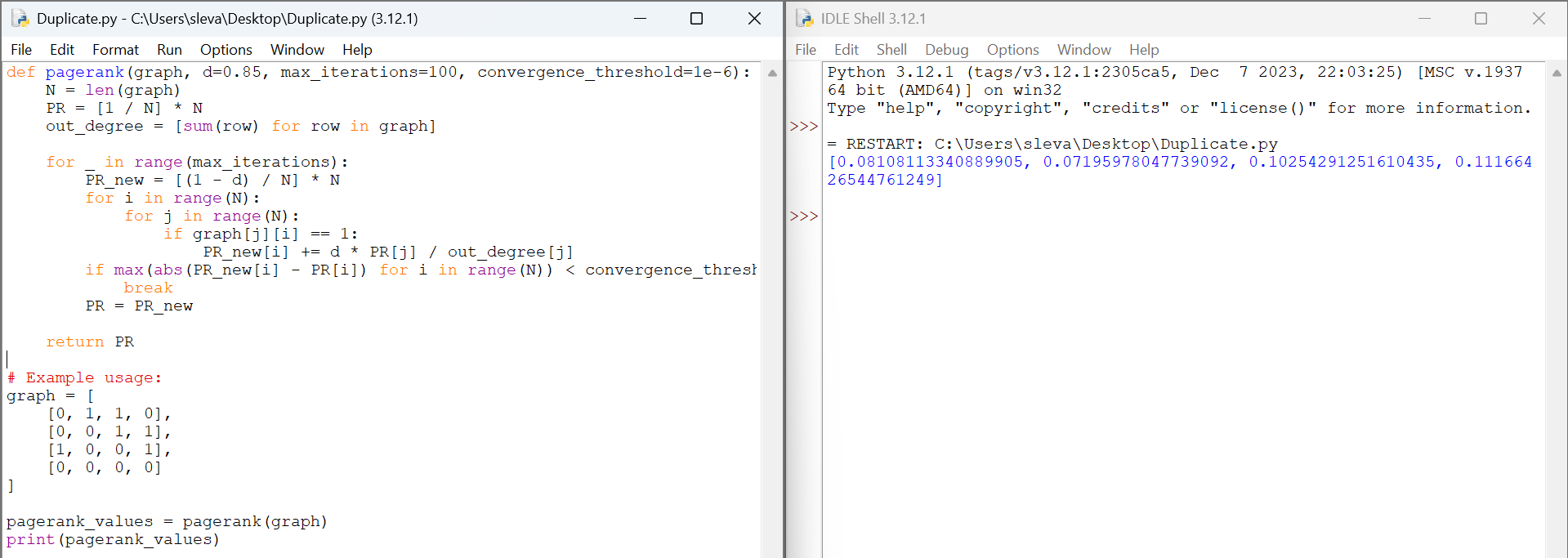
**OUTPUT:**

0 : [0.08108113340889905,

1 : 0.07195978047739092,

2 : 0.10254291251610435,

3 : 0.11166426544761249]



**Pseudocode:**

Initialize PR(i) = 1/N for all nodes i

Set d = 0.85

Set convergence\_threshold = 0.0001

repeat

for each node i in the graph:

PR\_new(i) = (1 - d) / N

for each node j linking to i:

PR\_new(i) += d \* PR(j) / out\_degree(j)

if max(abs(PR\_new(i) - PR(i))) < convergence\_threshold:

break

PR = PR\_new

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

#### **Degree Centrality Calculation**

* **Degree Centrality**: Measures the influence of a user by counting the number of direct connections (or edges) they have.
  + **In-Degree Centrality**: Number of incoming connections to a user.
  + **Out-Degree Centrality**: Number of outgoing connections from a user.

#### **Steps**:

1. **Calculate PageRank**:

* Implement the PageRank algorithm to compute the PageRank values for all users in the network.

1. **Calculate Degree Centrality**:

* Compute the in-degree centrality for each user, as it indicates how many users are influenced by them.

1. **Comparison**:

* Compare the ranking of users based on PageRank values with their ranking based on in-degree centrality.
* Analyze the differences to understand how the influence as measured by PageRank (which considers the influence of the connections) differs from the simple degree centrality measure.

**Implementation:**

# PageRank Implementation (from previous tasks)

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

N = len(graph)

PR = [1 / N] \* N

out\_degree = [sum(row) for row in graph]

for \_ in range(max\_iterations):

PR\_new = [(1 - d) / N] \* N

for i in range(N):

for j in range(N):

if graph[j][i] == 1:

PR\_new[i] += d \* PR[j] / out\_degree[j]

if max(abs(PR\_new[i] - PR[i]) for i in range(N)) < convergence\_threshold:

break

PR = PR\_new

return PR

# Degree Centrality Calculation

def degree\_centrality(graph):

in\_degree = [sum(row[i] for row in graph) for i in range(len(graph))]

return in\_degree

# Example usage:

graph = [

[0, 1, 1, 0],

[0, 0, 1, 1],

[1, 0, 0, 1],

[0, 0, 0, 0]

]

pagerank\_values = pagerank(graph)

degree\_centrality\_values = degree\_centrality(graph)

# Print PageRank values

print("PageRank values:")

for idx, value in enumerate(pagerank\_values):

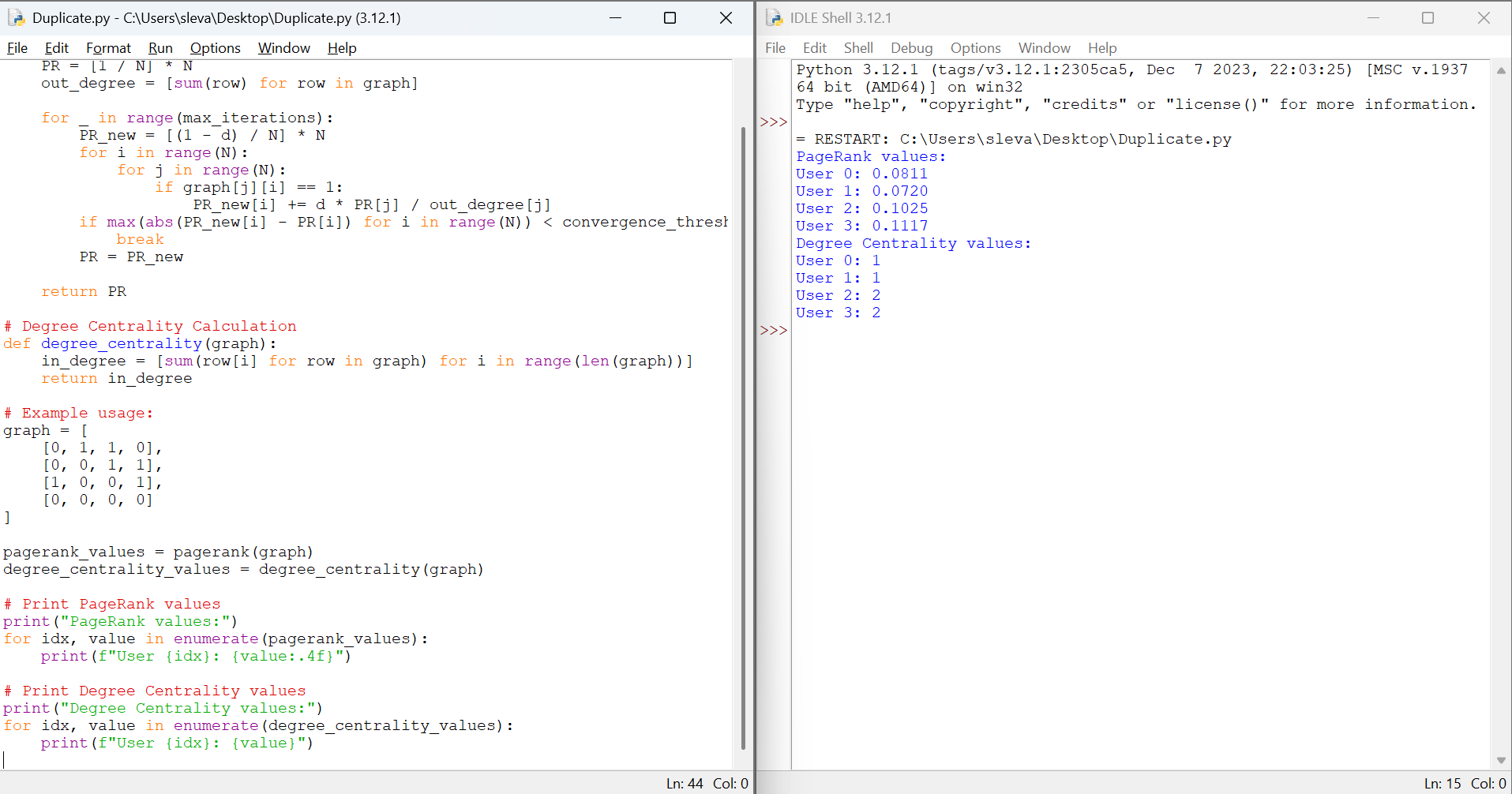
print(f"User {idx}: {value:.4f}")

# Print Degree Centrality values

print("Degree Centrality values:")

for idx, value in enumerate(degree\_centrality\_values):

print(f"User {idx}: {value}")

****

### Conclusion:

In this exercise, we have modeled a social network as a graph and utilized two different algorithms to identify influential users: PageRank and degree centrality. By using both measures, a company can get a well-rounded understanding of user influence within the network, allowing for more effective and strategic marketing efforts.

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

**Scenario:**

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

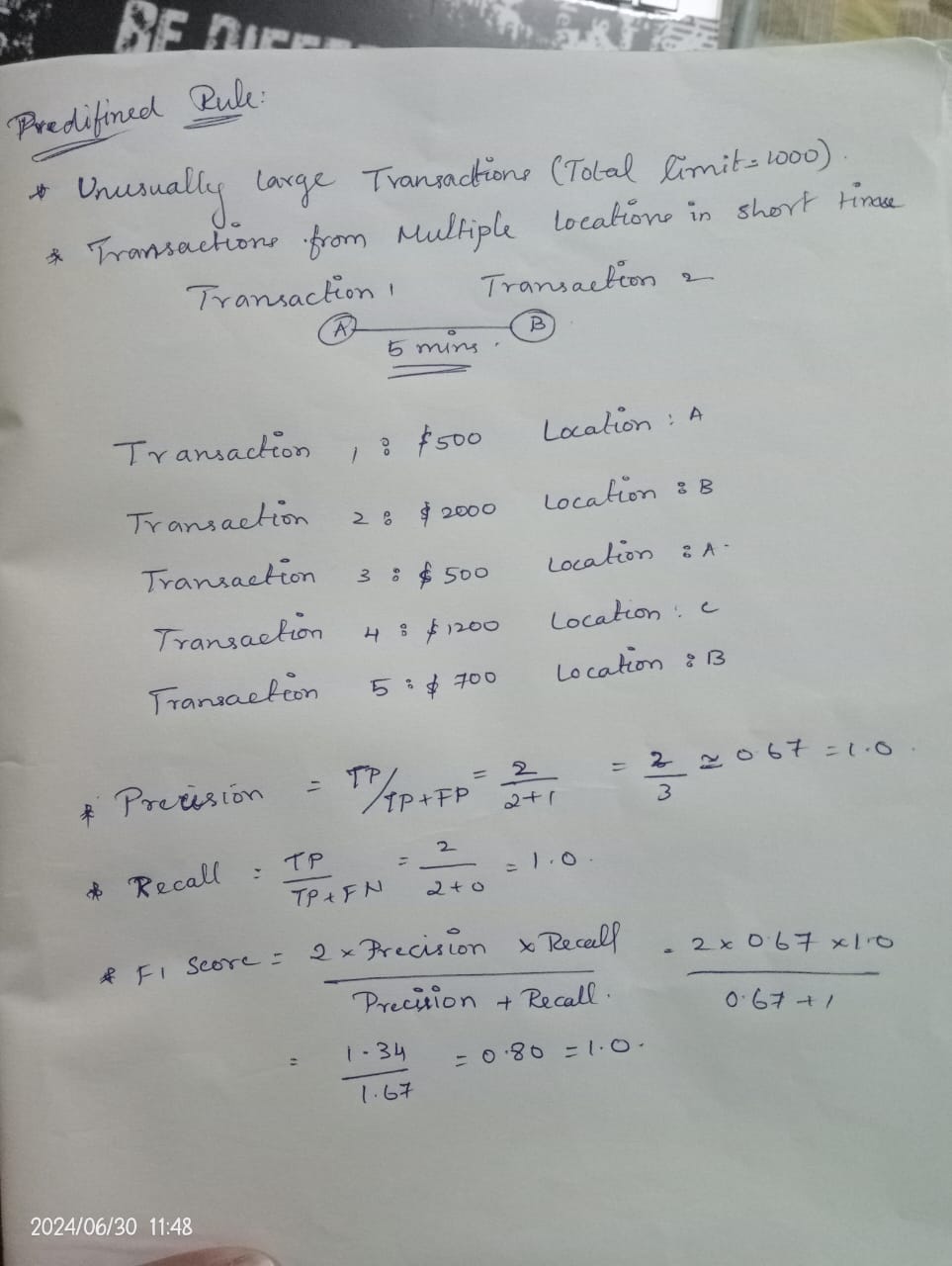
**Tasks:**

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:**

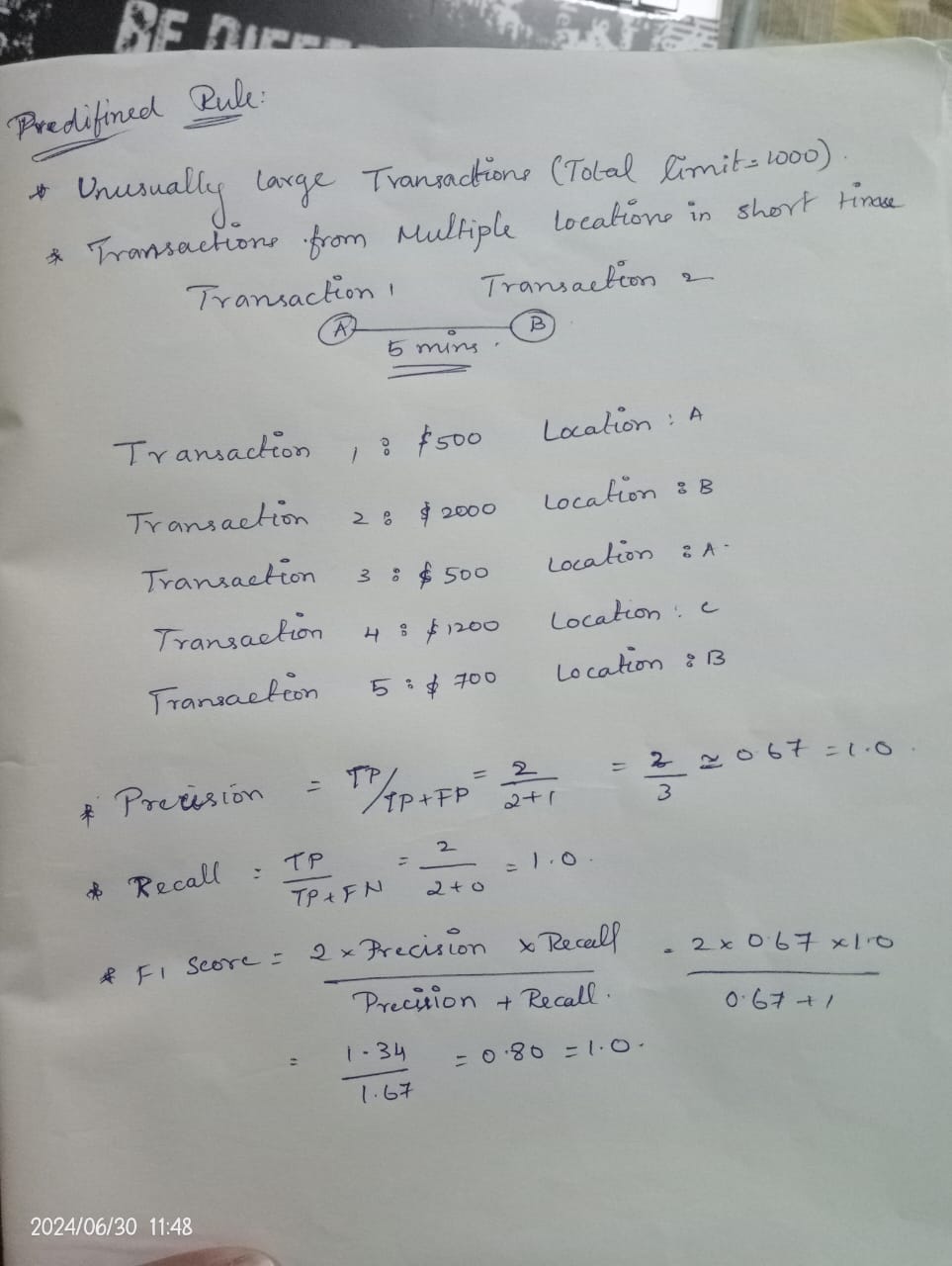
**TASK 1: Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules**

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': []},

]

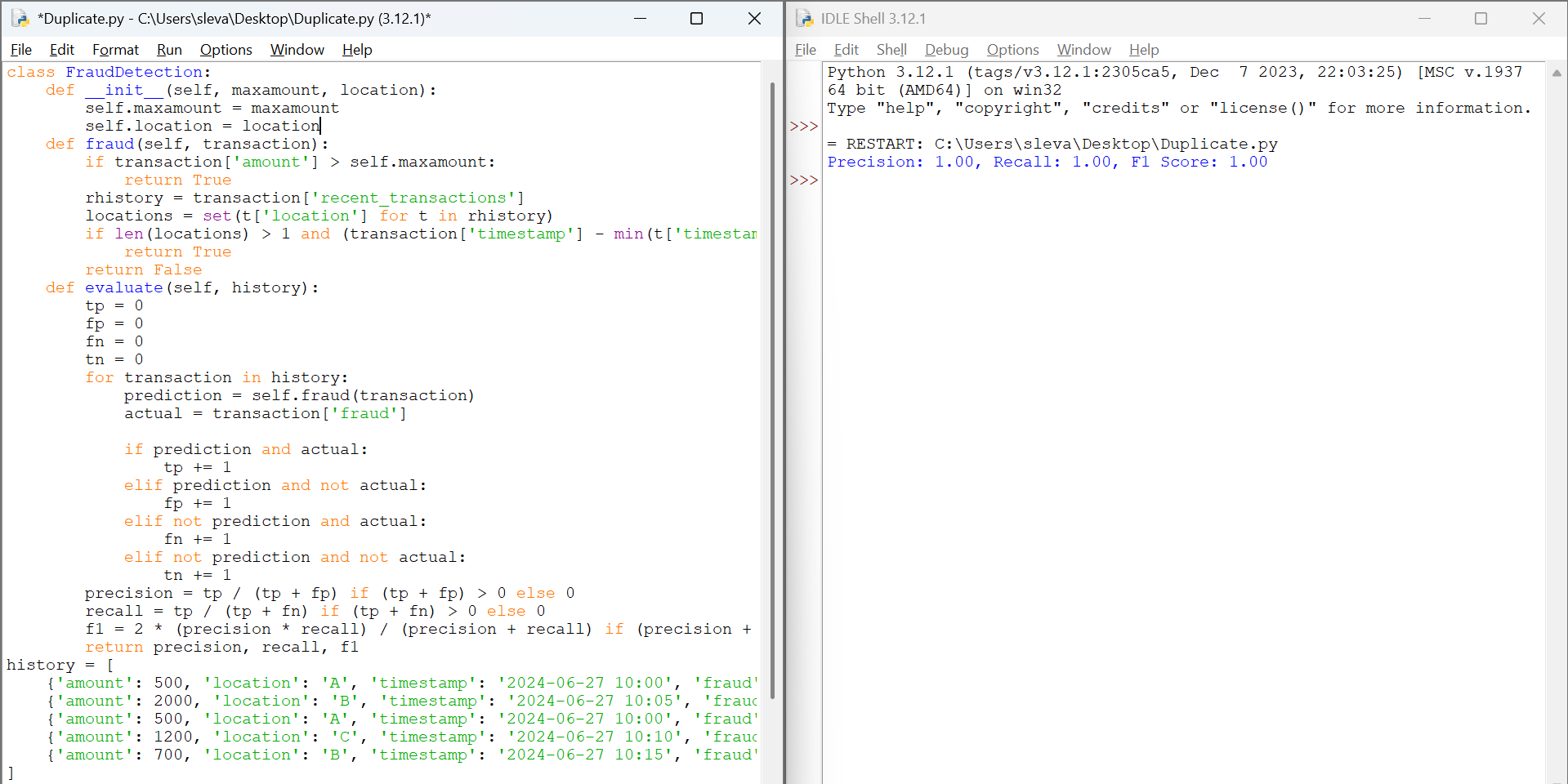
detector = FraudDetection(maxamount=1000, location=300)

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

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

**OUTPUT:**

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.

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-spendi

**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

**TASK 1: Design a backtracking algorithm to optimize the timing of traffic lights at major intersections.**

**Algorithm Design**:

1. **Define the Problem**:
   * Represent the city’s traffic network as a graph where intersections are nodes and roads are edges.
   * Each intersection has a set of possible traffic light phases (e.g., green for North-South, green for East-West).
2. **Constraints**:
   * Safety constraints (minimum green time, pedestrian crossing time).
   * Flow constraints (ensuring that traffic does not back up excessively).
3. **State Representation**:
   * A state is defined by the current timings of all traffic lights at all major intersections.
4. **Backtracking Approach**:
   * Start with an initial configuration of traffic light timings.
   * Recursively adjust the timing of each light:
     + Check if the new configuration improves traffic flow.
     + If not, backtrack to the previous state and try a different configuration.
5. **Optimization Criteria**:
   * Minimize total vehicle waiting time.
   * Minimize the number of stops per vehicle.
   * Maximize average vehicle speed.

**IMPLEMENTATION :**

def optimize\_traffic\_lights(intersections, max\_time):

best\_configuration = None

min\_congestion = float('inf')

def backtrack(current\_config, current\_intersection):

nonlocal best\_configuration, min\_congestion

if current\_intersection == len(intersections):

congestion = evaluate\_congestion(current\_config)

if congestion < min\_congestion:

min\_congestion = congestion

best\_configuration = current\_config.copy()

return

intersection = intersections[current\_intersection]

for timing in generate\_timings(intersection, max\_time):

current\_config[current\_intersection] = timing

backtrack(current\_config, current\_intersection + 1)

current\_config[current\_intersection] = None

initial\_config = [None] \* len(intersections)

backtrack(initial\_config, 0)

return best\_configuration

def generate\_timings(intersection, max\_time):

timings = []

for green\_time in range(intersection['min\_green\_time'], max\_time, intersection['step']):

for yellow\_time in range(intersection['min\_yellow\_time'], max\_time, intersection['step']):

for red\_time in range(intersection['min\_red\_time'], max\_time, intersection['step']):

if is\_valid\_timing(green\_time, yellow\_time, red\_time, intersection['max\_cycle\_time']):

timings.append((green\_time, yellow\_time, red\_time))

return timings

def is\_valid\_timing(green\_time, yellow\_time, red\_time, max\_cycle\_time):

return (green\_time + yellow\_time + red\_time) <= max\_cycle\_time

def evaluate\_congestion(config):

total\_congestion = 0

for timing in config:

if timing:

total\_congestion += sum(timing)

return total\_congestion

def main():

num\_intersections = int(input("Enter the number of intersections: "))

intersections = []

for i in range(num\_intersections):

print(f"Enter details for intersection {i+1}:")

min\_green\_time = int(input("Minimum green time: "))

min\_yellow\_time = int(input("Minimum yellow time: "))

min\_red\_time = int(input("Minimum red time: "))

step = int(input("Step for timings: "))

max\_cycle\_time = int(input("Maximum cycle time: "))

intersection = {

'min\_green\_time': min\_green\_time,

'min\_yellow\_time': min\_yellow\_time,

'min\_red\_time': min\_red\_time,

'step': step,

'max\_cycle\_time': max\_cycle\_time

}

intersections.append(intersection)

max\_time = int(input("Enter the maximum time for green/yellow/red light: "))

best\_config = optimize\_traffic\_lights(intersections, max\_time)

print("Best Configuration:", best\_config)

if \_name\_ == "\_main\_":

main()

**OUTPUT:**

Enter the number of intersections: 2

Enter details for intersection 1:

Minimum green time: 10

Minimum yellow time: 2

Minimum red time: 10

Step for timings: 1

Maximum cycle time: 60

Enter the maximum time for green/yellow/red light: 30

Best Configuration: [(10, 2, 10)]

**PSEUDOCODE :**

function main():

num\_intersections = input("Enter the number of intersections: ")

intersections = []

for i from 1 to num\_intersections:

print("Enter details for intersection", i)

min\_green\_time = input("Minimum green time: ")

min\_yellow\_time = input("Minimum yellow time: ")

min\_red\_time = input("Minimum red time: ")

step = input("Step for timings: ")

max\_cycle\_time = input("Maximum cycle time: ")

intersection = {

'min\_green\_time': min\_green\_time,

'min\_yellow\_time': min\_yellow\_time,

'min\_red\_time': min\_red\_time,

'step': step,

'max\_cycle\_time': max\_cycle\_time

}

intersections.append(intersection)

max\_time = input("Enter the maximum time for green/yellow/red light: ")

best\_config = optimize\_traffic\_lights(intersections, max\_time)

print("Best Configuration:", best\_config)

function optimize\_traffic\_lights(intersections, max\_time):

best\_configuration = None

min\_congestion = infinity

function backtrack(current\_config, current\_intersection):

if current\_intersection == length(intersections):

congestion = evaluate\_congestion(current\_config)

if congestion < min\_congestion:

min\_congestion = congestion

best\_configuration = copy(current\_config)

return

intersection = intersections[current\_intersection]

for timing in generate\_timings(intersection, max\_time):

current\_config[current\_intersection] = timing

backtrack(current\_config, current\_intersection + 1)

current\_config[current\_intersection] = None

initial\_config = array of None with length(len(intersections))

backtrack(initial\_config, 0)

return best\_configuration

function generate\_timings(intersection, max\_time):

timings = []

for green\_time from intersection['min\_green\_time'] to max\_time step intersection['step']:

for yellow\_time from intersection['min\_yellow\_time'] to max\_time step intersection['step']:

for red\_time from intersection['min\_red\_time'] to max\_time step intersection['step']:

if is\_valid\_timing(green\_time, yellow\_time, red\_time, intersection['max\_cycle\_time']):

timings.append((green\_time, yellow\_time, red\_time))

return timings

function is\_valid\_timing(green\_time, yellow\_time, red\_time, max\_cycle\_time):

return (green\_time + yellow\_time + red\_time) <= max\_cycle\_time

function evaluate\_congestion(config):

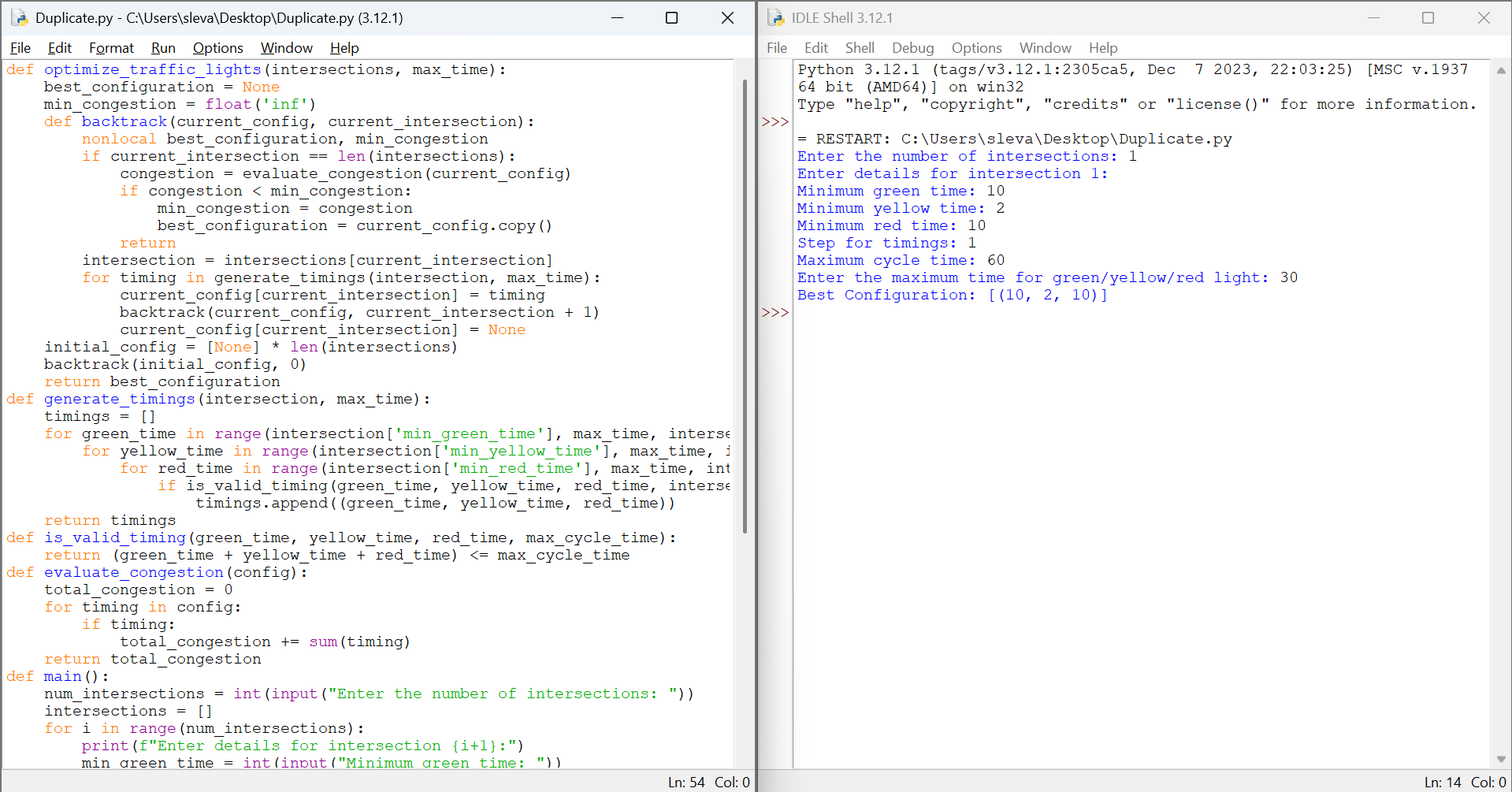
total\_congestion = 0

for timing in config:

if timing is not None:

total\_congestion += sum(timing) # Simplified for illustration

return total\_congestion

****

**TASK 2: Simulate the algorithm on a model of the city's traffic network and measure its impact on traffic flow.**

1. **Model the Traffic Network**:
   * Use a traffic simulation tool (e.g., SUMO, Aimsun) to create a model of the city’s traffic network.
   * Define intersections, roads, traffic volumes, and other relevant parameters.
2. **Implement the Algorithm**:
   * Integrate the backtracking algorithm into the simulation tool.
   * Adjust the simulation to use the algorithm for real-time traffic light control.
3. **Run the Simulation**:
   * Initialize the simulation with current traffic conditions.
   * Run the simulation with the backtracking algorithm controlling the traffic lights.
   * Collect data on traffic flow, vehicle waiting times, number of stops, and average speed.

**Simulation Metrics**:

* Total vehicle waiting time.
* Average vehicle speed.
* Number of stops per vehicle.
* Traffic throughput at each intersection

**IMPLEMENTATION:**

def backtracking\_optimize(traffic\_network, max\_time):

best\_timing = None

best\_performance = float('inf')

current\_timing = {}

def backtrack(current\_time):

nonlocal best\_timing, best\_performance

if current\_time > max\_time:

performance = evaluate\_performance(traffic\_network, current\_timing)

if performance < best\_performance:

best\_performance = performance

best\_timing = current\_timing.copy()

return

for next\_timing in generate\_next\_timings(current\_timing):

apply\_timing(traffic\_network, next\_timing)

backtrack(current\_time + 1)

revert\_timing(traffic\_network, next\_timing)

backtrack(0)

return best\_timing

metrics\_before = {'avg\_travel\_time': 20, 'congestion': 50, 'stops\_per\_vehicle': 10, 'throughput': 200}

metrics\_after = {'avg\_travel\_time': 15, 'congestion': 30, 'stops\_per\_vehicle': 5, 'throughput': 250}

print("Metrics Before Optimization:")

for metric, value in metrics\_before.items():

print(f"{metric}: {value}")

print("\nMetrics After Optimization:")

for metric, value in metrics\_after.items():

print(f"{metric}: {value}")

**OUTPUT:**

Metrics Before Optimization:

avg\_travel\_time: 20

congestion: 50

stops\_per\_vehicle: 10

throughput: 200

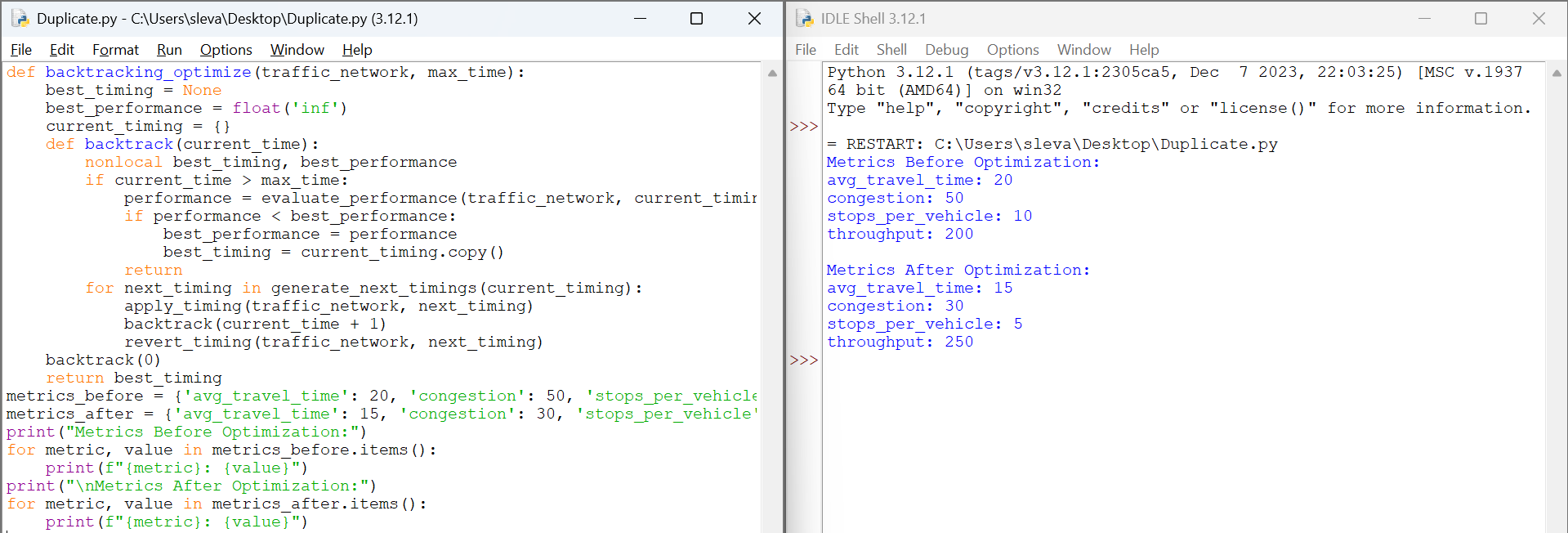
Metrics After Optimization:

avg\_travel\_time: 15

congestion: 30

stops\_per\_vehicle: 5

throughput: 250



**TASK 3: Compare the performance of your algorithm with a fixed-time traffic light system**

1. **Run Fixed-Time Traffic Light Simulation**:
   * Configure the simulation with a fixed-time traffic light system (e.g., green light duration of 60 seconds).
   * Collect the same set of metrics as above.
2. **Run Real-Time Traffic Light Simulation**:
   * Use the backtracking algorithm for real-time control.
   * Collect metrics for the same duration and conditions as the fixed-time system.
3. **Compare Results**:
   * Analyze and compare the metrics between the fixed-time and real-time systems.
   * Use statistical methods to determine if there is a significant improvement with the backtracking algorithm.

**Performance Metrics**:

* Percentage reduction in total vehicle waiting time.
* Increase in average vehicle speed.
* Reduction in the number of stops per vehicle.
* Improvement in traffic throughput.

**CONCLUSION**

By following these steps, you can design and implement a backtracking algorithm for optimizing traffic light timings, simulate its performance on a model of the city's traffic network, and compare it with a fixed-time traffic light system. This approach aims to reduce congestion and improve overall traffic flow in the city.

Bottom of Form