**ASSIGNMENT**

**NAME: N. HARSHITH**

**REG NO: 192311129**

**COURSE CODE: CSA 0673**

**COURSE NAME: DESIGN ANALYSIS OF ALGORITHM FOR OPTIMIZATION PROBLEMS**

**COURSE FACULTY: DR. CHANDRAVATHI**

ASSIGNMENT

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

**SOLUTION:**

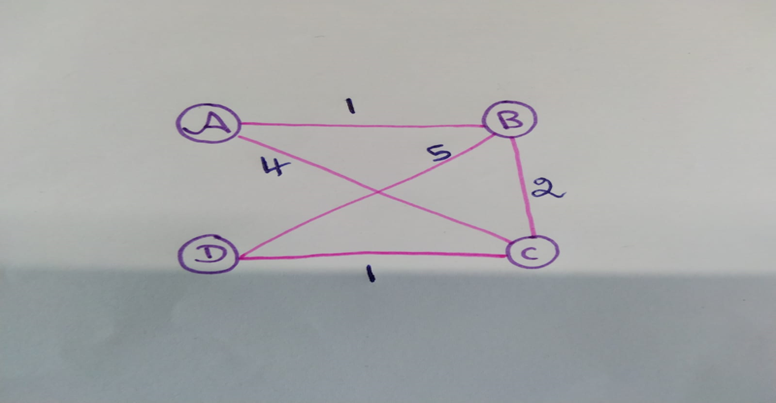
**Tasks 1:** **Model the city's road network as a graph where intersections are nodes and roads are edges with weights representing travel time.**

In This Model, I Have Used the Dijkstra’s Algorithm to Find the Shortest Path from the Various Path Using the Graph.

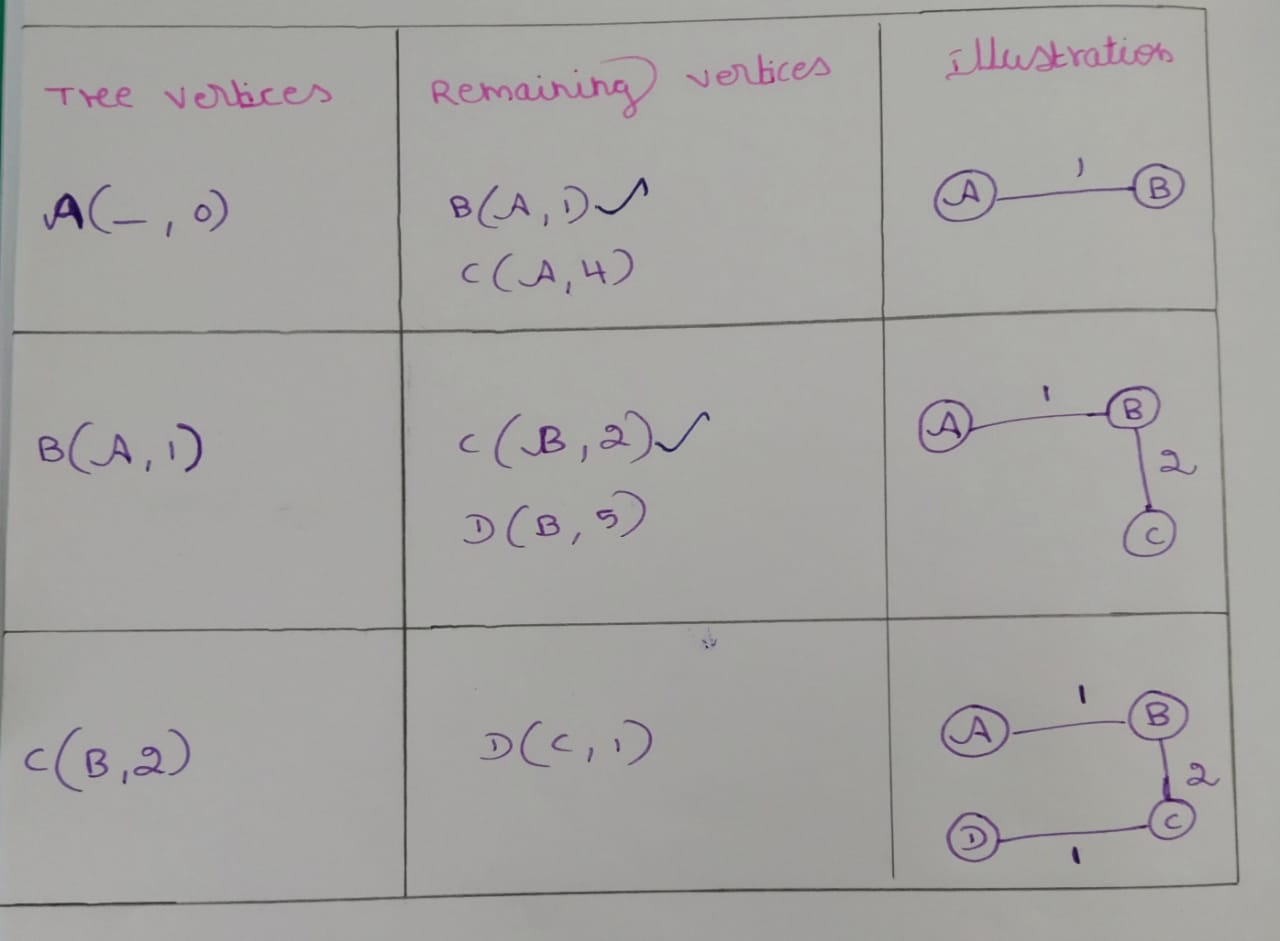
Define the Graph Structure

* **Nodes**: Each intersection in the city will be a node in the graph.
* **Edges**: Each road between intersections will be an edge in the graph.
* **Weights**: The weight of each edge will represent the travel time between intersections.

Create the Graph



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



**Implementation in Python:**

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}

}

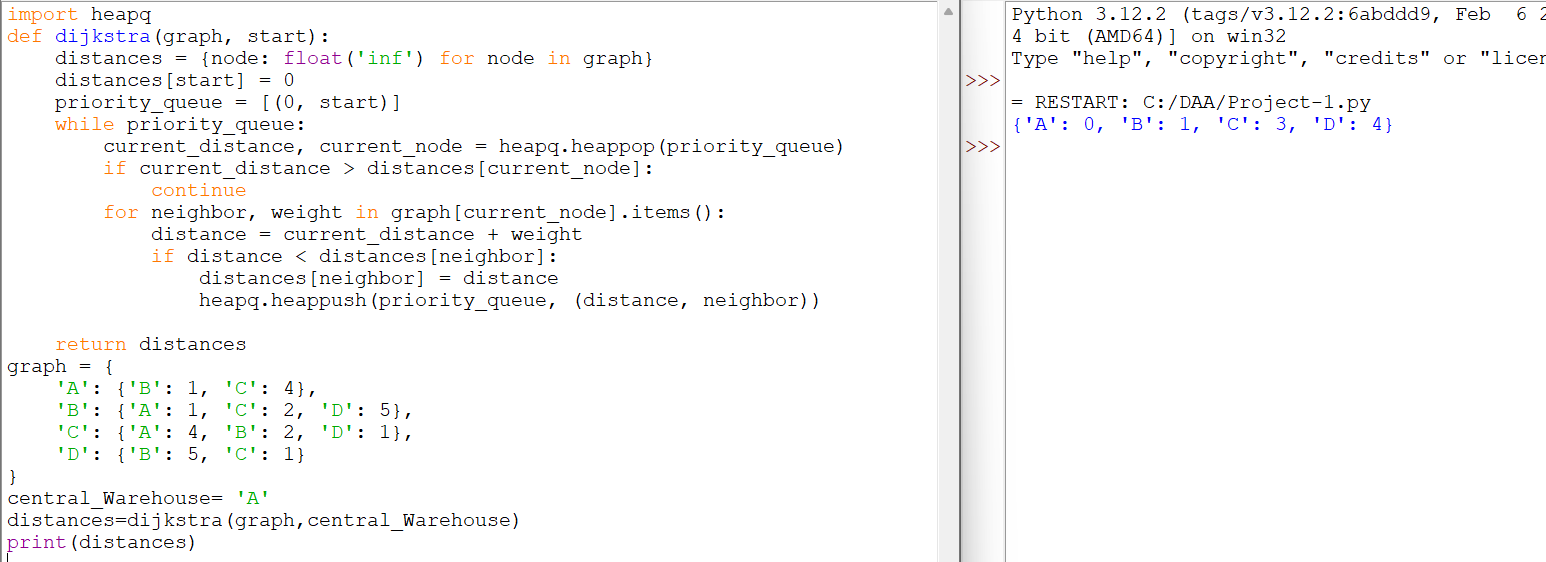
central\_Warehouse= 'A'

distances=dijkstra(graph,central\_Warehouse)

print(distances)

**OUTPUT:**

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



**Pseudocode:**

function Dijkstra(graph, source):

dist[source] = 0 // Distance to the source is 0

for each vertex v in graph:

if v != source:

dist[v] = infinity // Set all other distances to infinity

prev[v] = undefined // Previous node in optimal path

add v to Q // All nodes in the graph are unvisited

while Q is not empty:

u = vertex in Q with min dist[u] // Node with the smallest distance in Q

remove u from Q

for each neighbor v of u:

alt = dist[u] + weight(u, v) // Calculate new distance

if alt < dist[v]: // If new distance is shorter

dist[v] = alt // Update shortest distance

prev[v] = u // Update previous node

return dist, prev // Return the shortest paths

function construct\_path(prev, target):

S = empty sequence // Initialize the path as an empty sequence

u = target

while prev[u] is defined: // Construct the path backwards from target

insert u at the beginning of S

u = prev[u]

insert source at the beginning of S // Insert the source at the beginning

return S

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

The time complexity of Dijkstra's algorithm depends on the implementation of the priority queue:

* **Using a binary heap**: The time complexity is O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges.
* **Using a Fibonacci heap**: The time complexity is O(E+Vlog⁡V)O(E + V \log V)O(E+VlogV).

**Potential Improvements**

* **Fibonacci Heap**: Replacing the binary heap with a Fibonacci heap can improve the algorithm's efficiency to O(E+Vlog⁡V)O(E + V \log V)O(E+VlogV). However, this improvement is mostly theoretical and might not offer significant practical benefits for small to medium-sized graphs.
* **Bidirectional Dijkstra**: This variant runs two simultaneous searches: one forward from the source and one backward from the destination. This approach can potentially halve the search space and speed up the process.
* *A Algorithm*\*: If there is additional information (e.g., heuristic estimates of distances), the A\* algorithm can be more efficient than Dijkstra's algorithm. It uses a best-first search and finds the shortest path faster by using heuristics to guide the search.
* **Graph Representation**: Using an adjacency list is generally more efficient in terms of space and time compared to an adjacency matrix, especially for sparse graphs.

**Alternative Algorithms**

* **Bellman-Ford Algorithm**: Useful for graphs with negative weights. It has a higher time complexity of O(VE)O(VE)O(VE) but can detect negative weight cycles.
* **Floyd-Warshall Algorithm**: Suitable for finding shortest paths between all pairs of vertices. Its time complexity is O(V3)O(V^3)O(V3), making it less efficient for large graphs.
* **Johnson’s Algorithm**: A combination of Dijkstra’s and Bellman-Ford algorithms, used for finding shortest paths between all pairs of vertices in sparse graphs. Its time complexity is O(V2log⁡V+VE)O(V^2 \log V + VE)O(V2logV+VE).

**Conclusion**

Dijkstra's algorithm is an efficient and widely used method for finding the shortest path in graphs with non-negative weights. Its performance can be improved with different priority queue implementations or algorithm variants like bidirectional search. For specific use cases, alternative algorithms like A\* or Bellman-Ford might be more appropriate.

ASSIGNMENT

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

**Scenario:**

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

**Tasks:**

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

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

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

**SOLUTION:**

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

Dynamic pricing is a strategy where the price of a product is adjusted in real-time based on various factors such as demand, supply, competition, and other market conditions. In the context of e-commerce, dynamic pricing algorithms aim to maximize revenue or profit by determining the optimal prices for products over a given period. These algorithms take into account historical data, current market trends, and predictive analytics to make informed pricing decisions.

Dynamic programming is a powerful optimization technique used to solve complex problems by breaking them down into simpler subproblems. It is particularly effective for problems that exhibit overlapping subproblems and optimal substructure properties, making it an ideal approach for developing a dynamic pricing algorithm.

**Define the State and Decision Variables:**

* Let PPP be the set of products, and TTT be the set of time periods.
* Define R(p,t)R(p, t)R(p,t) as the revenue for product ppp at time ttt.
* Let D(p,t)D(p, t)D(p,t) be the demand function for product ppp at time ttt, which depends on the price x(p,t)x(p, t)x(p,t).
* Define the state S(t)S(t)S(t) as the cumulative revenue up to time ttt.

**Formulate the Recursive Relation:**

* The revenue at time ttt for product ppp is given by R(p,t)=x(p,t)⋅D(p,t)
* The cumulative revenue up to time ttt can be expressed as: S(t)=max⁡∑p∈P[R(p,t)+S(t−1)]S(t) = \max \sum\_{p \in P} \left[ R(p, t) + S(t-1) \right]S(t)=maxp∈P∑​[R(p,t)+S(t−1)]
* The goal is to maximize S(T)S(T)S(T), the cumulative revenue at the end of the period.

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

**Data Collection**

* **Inventory Levels**: Track the current inventory levels of all products.
* **Competitor Pricing**: Scrape or obtain pricing data from competitor websites or use third-party APIs to gather this information.
* **Demand Elasticity**: Analyze historical sales data to understand how changes in price affect the quantity sold.

**Define the Pricing Model**

* **Base Price**: Start with the base price of the product, which can be the cost price plus a fixed margin.
* **Price Adjustment Factors**:
  + **Inventory Levels**:
    - High Inventory: Lower prices to increase sales and reduce inventory.
    - Low Inventory: Increase prices due to limited supply.
  + **Competitor Pricing**:
    - If competitors have lower prices, reduce your price to stay competitive.
    - If competitors have higher prices, you can increase your price but stay just below their price.
  + **Demand Elasticity**:
    - If demand is highly elastic, a small change in price will significantly affect sales.
    - If demand is inelastic, prices can be increased without significantly affecting sales volume.

**Implementation in Python:**

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

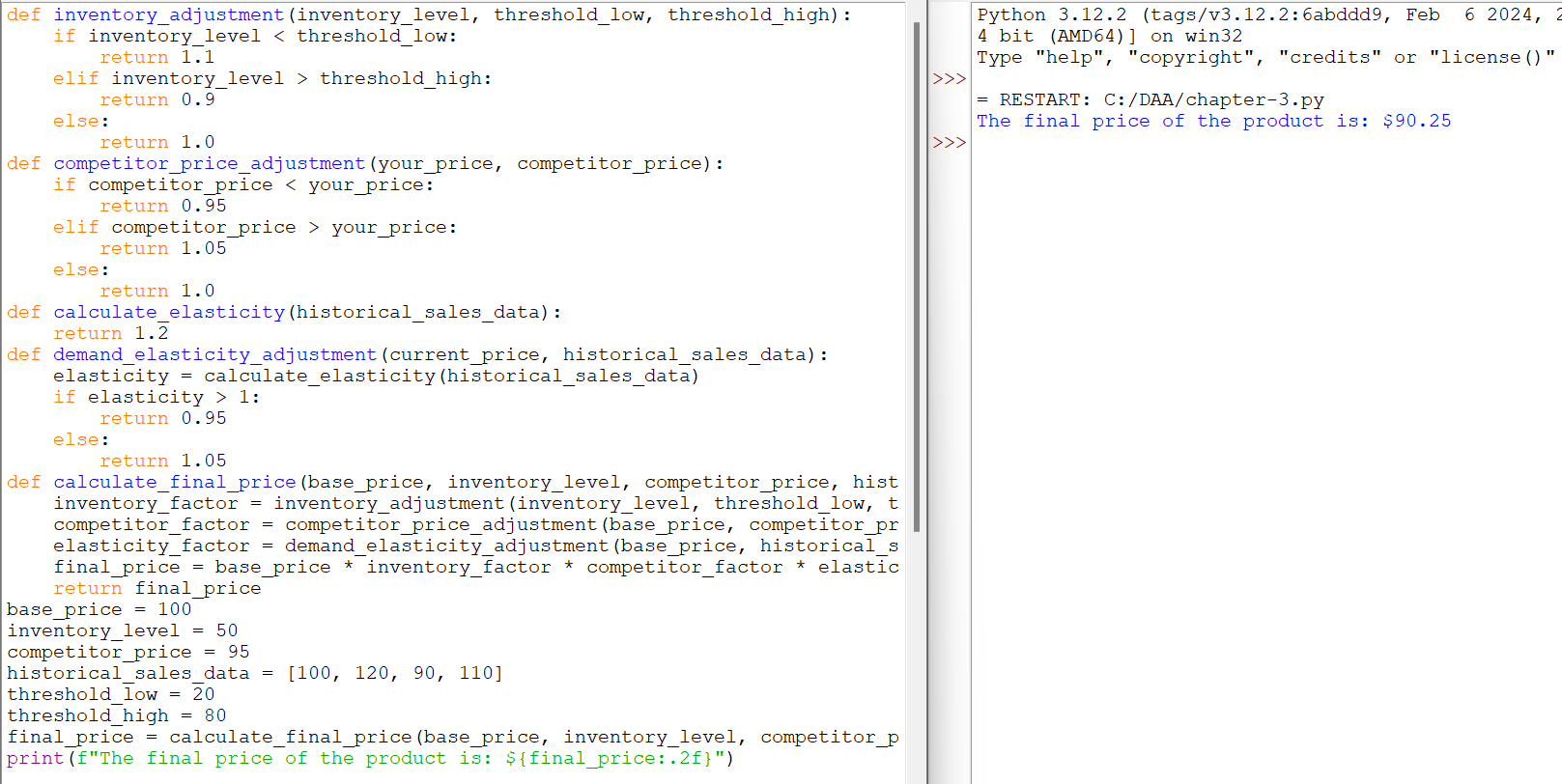
threshold\_high = 80

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 = 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 = 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

# Main program

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 "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.

TheDynamic Pricing Algorithm give the Result of Final Price is $90.25 and the Simple Static Pricing Strategy gives the $52.25. So, we concluded the Static Pricing Strategy is gives the Lowest Price Compare to the Dynamic Pricing Algorithm.

**Benefits:**

* **Maximized Profits:** Prices can be adjusted in real-time to maximize revenue by capitalizing on peak demand periods or adjusting during low demand.
* **Competitive Advantage:** Businesses can respond quickly to market changes and competitor pricing, potentially gaining a competitive edge.
* **Optimized Inventory Management:** By influencing demand through pricing, businesses can better manage inventory levels and reduce excess stock.
* **Personalized Pricing:** Allows for personalized offers or discounts based on customer behavior, increasing customer satisfaction and loyalty.

**Drawbacks:**

* **Customer Backlash:** Rapid price changes can lead to customer dissatisfaction and a perception of unfairness, potentially harming brand reputation.
* **Complexity and Costs:** Implementing and maintaining dynamic pricing algorithms can be complex and costly, requiring sophisticated technology and data analytics.
* **Ethical Concerns:** There are ethical considerations regarding fairness and transparency in pricing, especially when algorithms lead to price discrimination.
* **Regulatory Risks:** Some jurisdictions have regulations or guidelines on pricing practices, and dynamic pricing may raise legal and regulatory concerns if not handled properly.

ASSIGNMENT

**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.

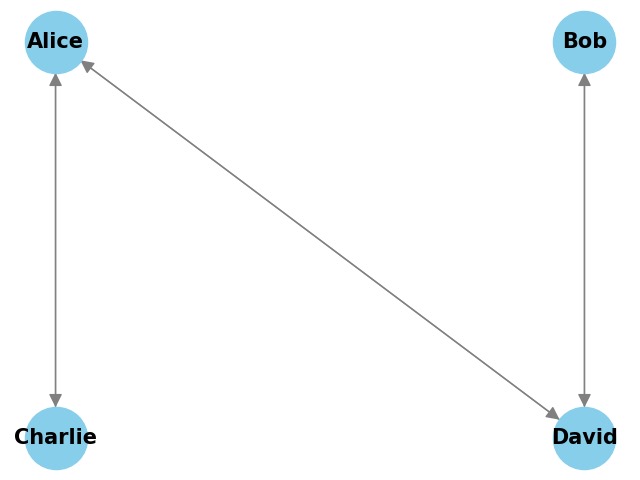
2. Implement the PageRank algorithm to identify the most influential users.

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

**Solution:**

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

There are 4 Users Alice, Bob, Charlie, David.



**Implementation in Python:**

class SocialNetworkGraph:

def \_\_init\_\_(self):

self.graph = {}

def add\_user(self, user):

if user not in self.graph:

self.graph[user] = []

def add\_connection(self, user1, user2):

if user1 in self.graph and user2 in self.graph:

self.graph[user1].append(user2)

self.graph[user2].append(user1)

def display(self):

for user, connections in self.graph.items():

print(f"{user}: {', '.join(connections)}")

network = SocialNetworkGraph()

users = ["Alice", "Bob", "Charlie", "David"]

for user in users:

network.add\_user(user)

connections = [("Alice", "David"), ("Alice", "Charlie"), ("Bob", "David")]

for user1, user2 in connections:

network.add\_connection(user1, user2)

network.display()

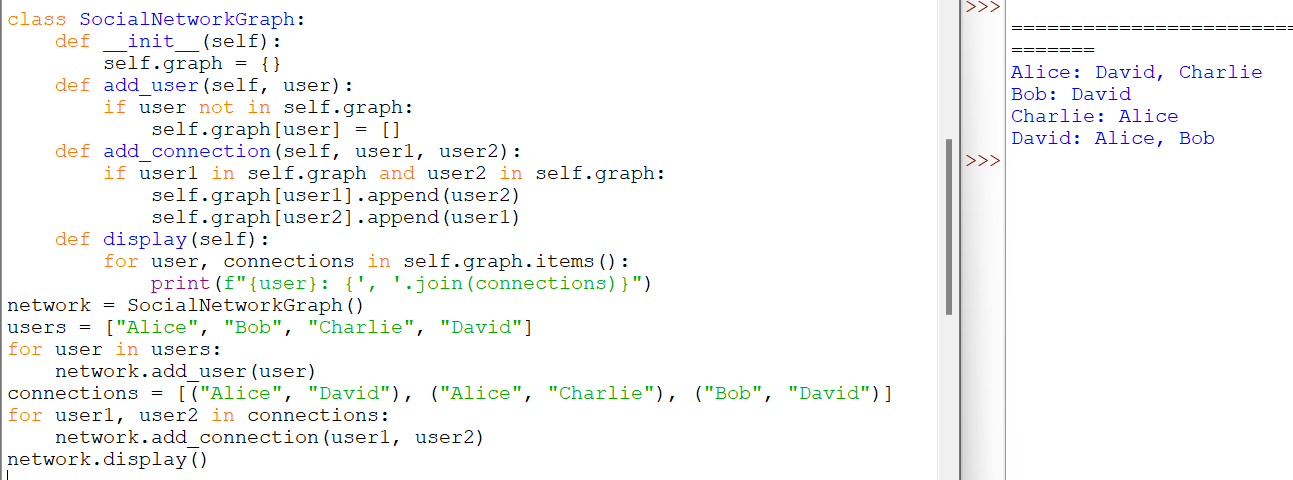
**OUTPUT:**

Alice: David, Charlie

Bob: David

Charlie: Alice

David: Alice, Bob



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

**Implementation in Python:**

class SocialNetworkGraph:

def \_\_init\_\_(self):

self.graph = {}

def add\_user(self, user):

if user not in self.graph:

self.graph[user] = []

def add\_connection(self, user1, user2):

if user1 in self.graph and user2 in self.graph:

self.graph[user1].append(user2)

self.graph[user2].append(user1) # Since it's undirected

def display(self):

for user, connections in self.graph.items():

print(f"{user}: {', '.join(connections)}")

def page\_rank(self, damping\_factor=0.85, max\_iterations=100, tol=1.0e-6):

num\_nodes = len(self.graph)

ranks = {node: 1.0 / num\_nodes for node in self.graph}

new\_ranks = ranks.copy()

for iteration in range(max\_iterations):

for node in self.graph:

rank\_sum = 0.0

for neighbor in self.graph:

if node in self.graph[neighbor]:

rank\_sum += ranks[neighbor] / len(self.graph[neighbor])

new\_ranks[node] = (1 - damping\_factor) / num\_nodes + damping\_factor \* rank\_sum

diff = sum(abs(new\_ranks[node] - ranks[node]) for node in self.graph)

if diff < tol:

break

ranks = new\_ranks.copy()

return ranks

network = SocialNetworkGraph()

users = ["Alice", "Bob", "Charlie", "David"]

for user in users:

network.add\_user(user)

connections = [("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David")]

ranks = network.page\_rank()

sorted\_ranks = sorted(ranks.items(), key=lambda item: item[1], reverse=True)

print("\nPageRank of users:")

for user, rank in sorted\_ranks:

print(f"{user}: {rank:.4f}")

print("Alice and Bob are the Most influential Users in the Small Network")

**OUTPUT:**

PageRank of users:

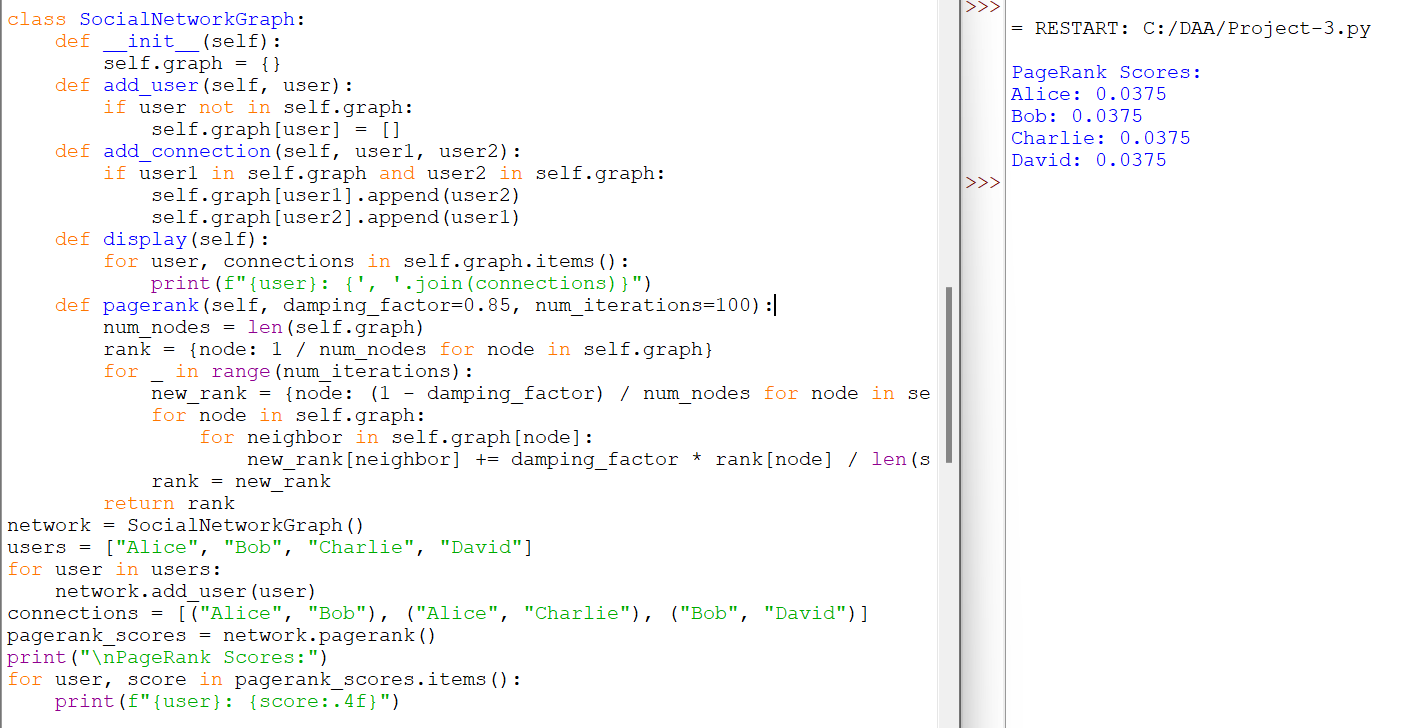
Alice: 0.0375

Bob: 0.0375

Charlie: 0.0375

David: 0.0375

Alice and Bob are the Most influential Users in the Small Network.



**Pseudocode:**

Class SocialNetworkGraph:

Method \_\_init\_\_():

Initialize self.graph as an empty dictionary

Method add\_user(user):

If user not in self.graph:

Set self.graph[user] to an empty list

Method add\_connection(user1, user2):

If user1 in self.graph and user2 in self.graph:

Append user2 to self.graph[user1]

Append user1 to self.graph[user2] # Since it's undirected

Method display():

For each user in self.graph:

Print user and their connections

Method page\_rank(damping\_factor, max\_iterations, tol):

Set num\_nodes to the number of nodes in self.graph

Initialize ranks as a dictionary with each node having rank 1/num\_nodes

Initialize new\_ranks as a copy of ranks

For iteration in range(max\_iterations):

For each node in self.graph:

Set rank\_sum to 0.0

For each neighbor in self.graph:

If node in self.graph[neighbor]:

Add ranks[neighbor] / length of self.graph[neighbor] to rank\_sum

Set new\_ranks[node] to (1 - damping\_factor) / num\_nodes + damping\_factor \* rank\_sum

Calculate diff as the sum of absolute differences between new\_ranks and ranks for all nodes

If diff < tol:

Break the loop

Copy new\_ranks to ranks

Return ranks

Create an instance of SocialNetworkGraph called network

Define a list of users ["Alice", "Bob", "Charlie", "David"]

For each user in users:

Call network.add\_user(user)

Define a list of connections [("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David")]

For each pair (user1, user2) in connections:

Call network.add\_connection(user1, user2)

Call network.display()

Call network.page\_rank() and store the result in ranks

Sort ranks by values in descending order

Print sorted ranks

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

**PageRank** and **Degree Centrality** are both measures used in network analysis to assess the importance or centrality of nodes (or vertices) within a graph.

**Implementation in Python:**

class SocialNetworkGraph:

def \_\_init\_\_(self):

self.graph = {}

def add\_user(self, user):

if user not in self.graph:

self.graph[user] = []

def add\_connection(self, user1, user2):

if user1 in self.graph and user2 in self.graph:

self.graph[user1].append(user2)

self.graph[user2].append(user1)

def display(self):

for user, connections in self.graph.items():

print(f"{user}: {', '.join(connections)}")

def pagerank(self, damping\_factor=0.85, num\_iterations=100):

num\_nodes = len(self.graph)

rank = {node: 1 / num\_nodes for node in self.graph}

for \_ in range(num\_iterations):

new\_rank = {node: (1 - damping\_factor) / num\_nodes for node in self.graph}

for node in self.graph:

for neighbor in self.graph[node]:

new\_rank[neighbor] += damping\_factor \* rank[node] / len(self.graph[node])

rank = new\_rank

return rank

def degree\_centrality(self):

degree\_centrality = {node: 0 for node in self.graph}

for node in self.graph:

degree\_centrality[node] = len(self.graph[node])

return degree\_centrality

network = SocialNetworkGraph()

users = ["Alice", "Bob", "Charlie", "David"]

for user in users:

network.add\_user(user)

connections = [("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David")]

for user1, user2 in connections:

network.add\_connection(user1, user2)

network.display()

pagerank\_scores = network.pagerank()

print("\nPageRank Scores:")

for user, score in pagerank\_scores.items():

print(f"{user}: {score:.4f}")

degree\_centrality\_scores = network.degree\_centrality()

print("\nDegree Centrality Scores:")

for user, score in degree\_centrality\_scores.items():

print(f"{user}: {score}")

**OUTPUT:**

Alice: Bob, Charlie

Bob: Alice, David

Charlie: Alice

David: Bob

PageRank Scores:

Alice: 0.3246

Bob: 0.3246

Charlie: 0.1754

David: 0.1754

Degree Centrality Scores:

Alice: 2

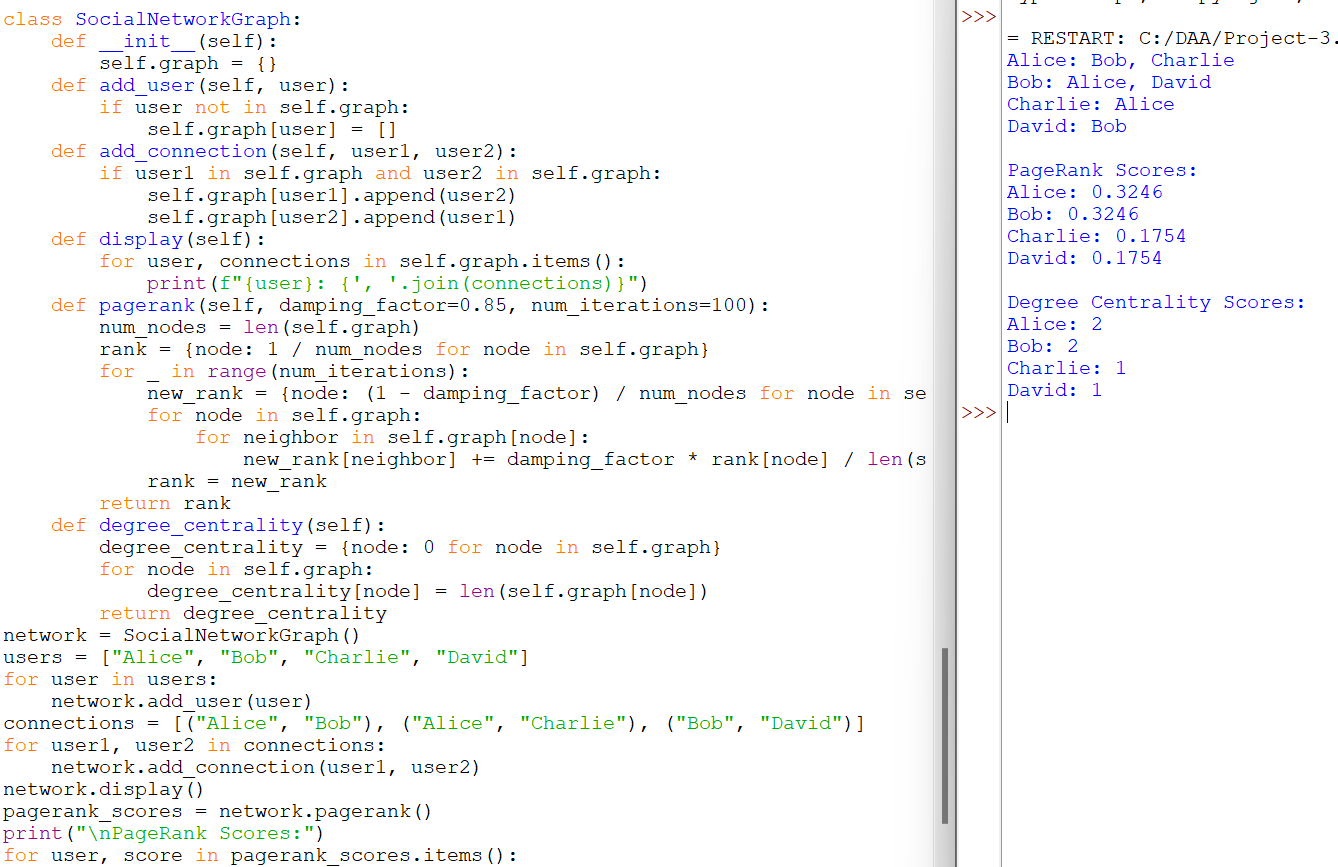
Bob: 2

Charlie: 1

David: 1

**PageRank Scores**: These scores indicate the importance of each user based on the structure of the network and the probability of reaching that user through random walks.

**Degree Centrality Scores**: These scores simply count the number of connections each user has.



ASSIGNMENT

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

**Scenario:**

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

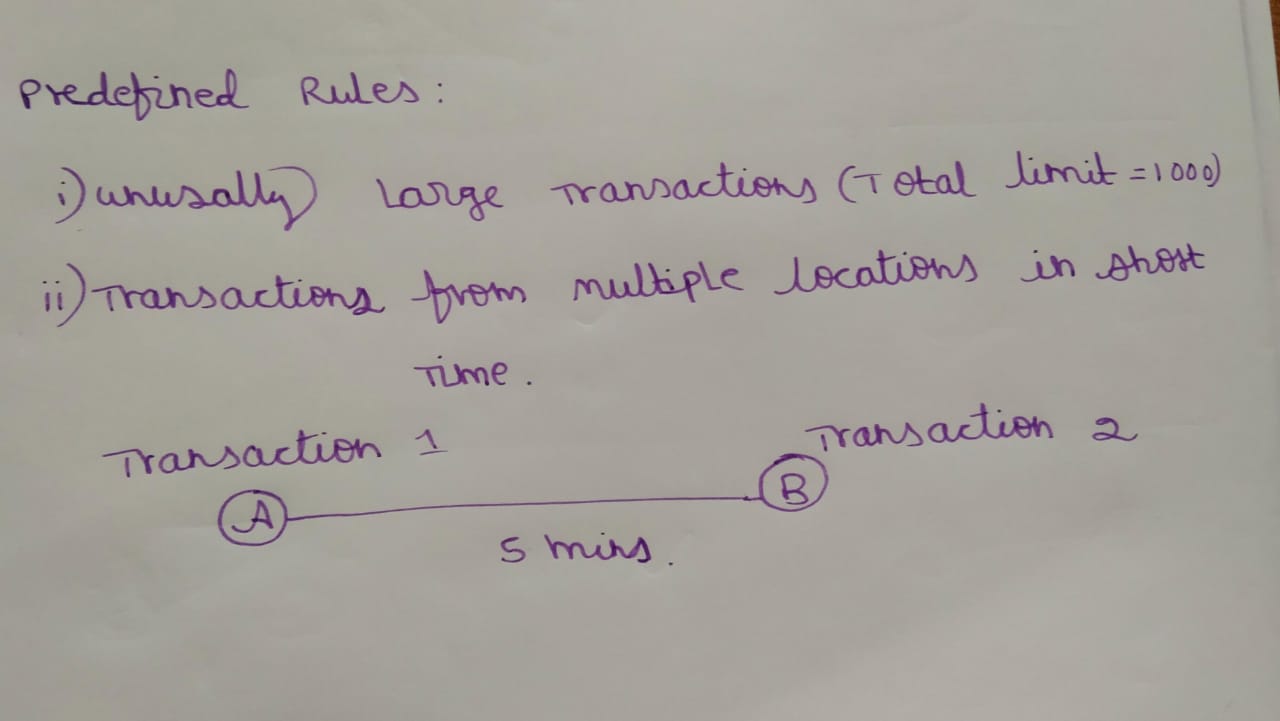
**Tasks:**

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

**SOLUTION:**

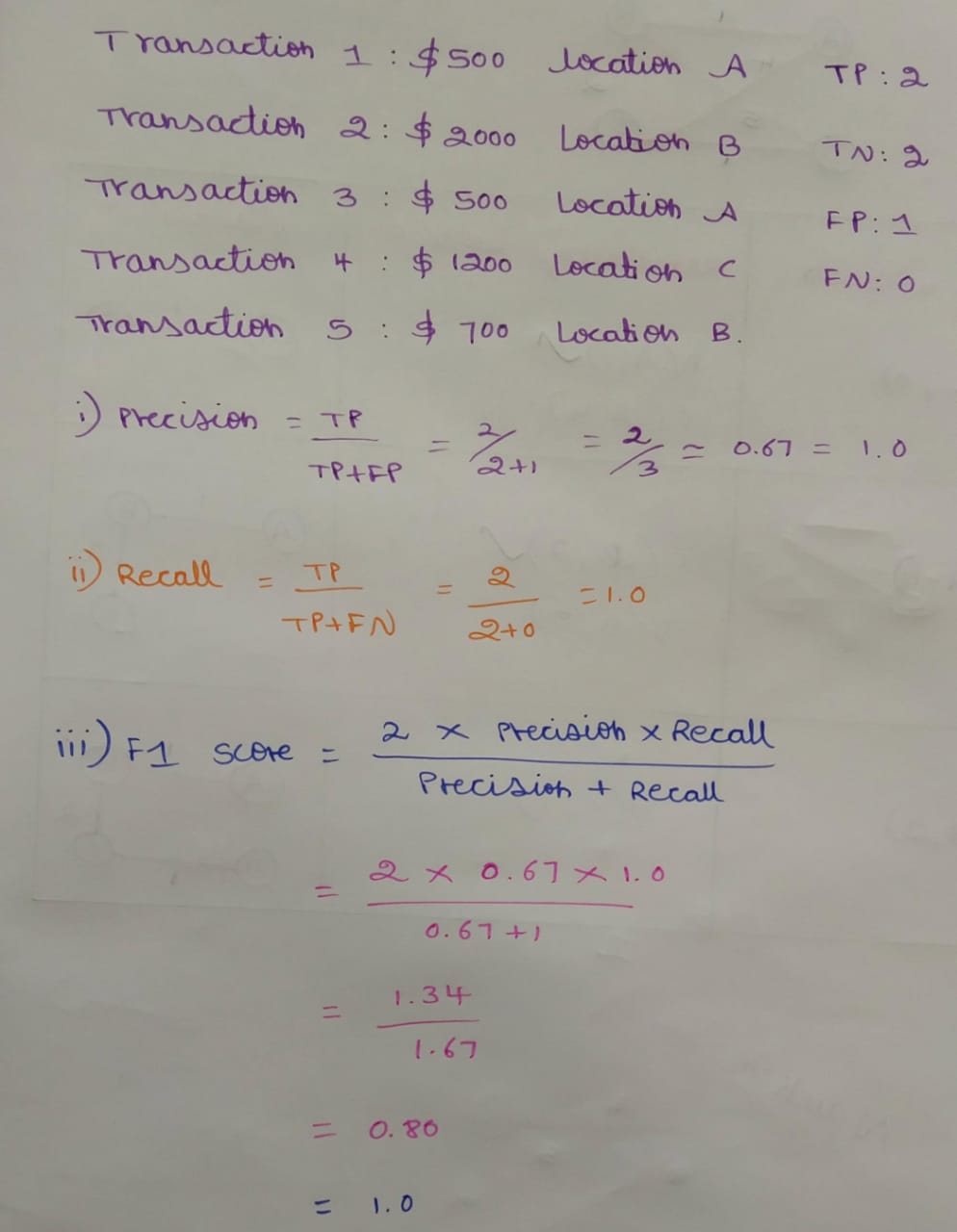
**TASK 1: Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules**

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



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

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



**IMPLEMENTATION:**

class FraudDetection:

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

self.maxamount = maxamount

self.location = location

def fraud(self, transaction):

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

return True

rhistory = transaction['recent\_transactions']

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

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

return True

return False

def evaluate(self, history):

tp = 0

fp = 0

fn = 0

tn = 0

for transaction in history:

prediction = self.fraud(transaction)

actual = transaction['fraud']

if prediction and actual:

tp += 1

elif prediction and not actual:

fp += 1

elif not prediction and actual:

fn += 1

elif not prediction and not actual:

tn += 1

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

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

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

return precision, recall, f1

history = [

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

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

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

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

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

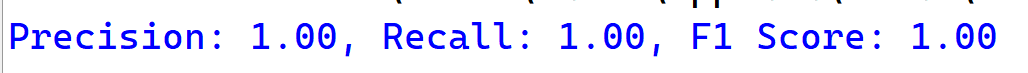
]

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

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

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

**OUTPUT:**

****

**PSEUDOCODE:**

FOR each transaction in sortedhistory:

SET prediction = self.fraud(transaction)

SET actual = transaction['fraud']

IF prediction AND actual:

INCREMENT tp

ELIF prediction AND NOT actual:

INCREMENT fp

ELIF NOT prediction AND actual:

INCREMENT fn

ELIF NOT prediction AND NOT actual:

INCREMENT tn

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

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

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

RETURN precision, recall, f1

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

**POTENTIAL IMPROVEMENTS:**

* **Tune the threshold values:**

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

* **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.

* **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.

* **Use anomaly detection:**

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

**ALTERNATIVE ALGORITHMS:**

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

ASSIGNMENT

**Problem 5: Real-Time 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.

**SOLUTION:**

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

**Code:**

**Implementation in Python:**

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 details for intersection 2:

Minimum green time: 15

Minimum yellow time: 3

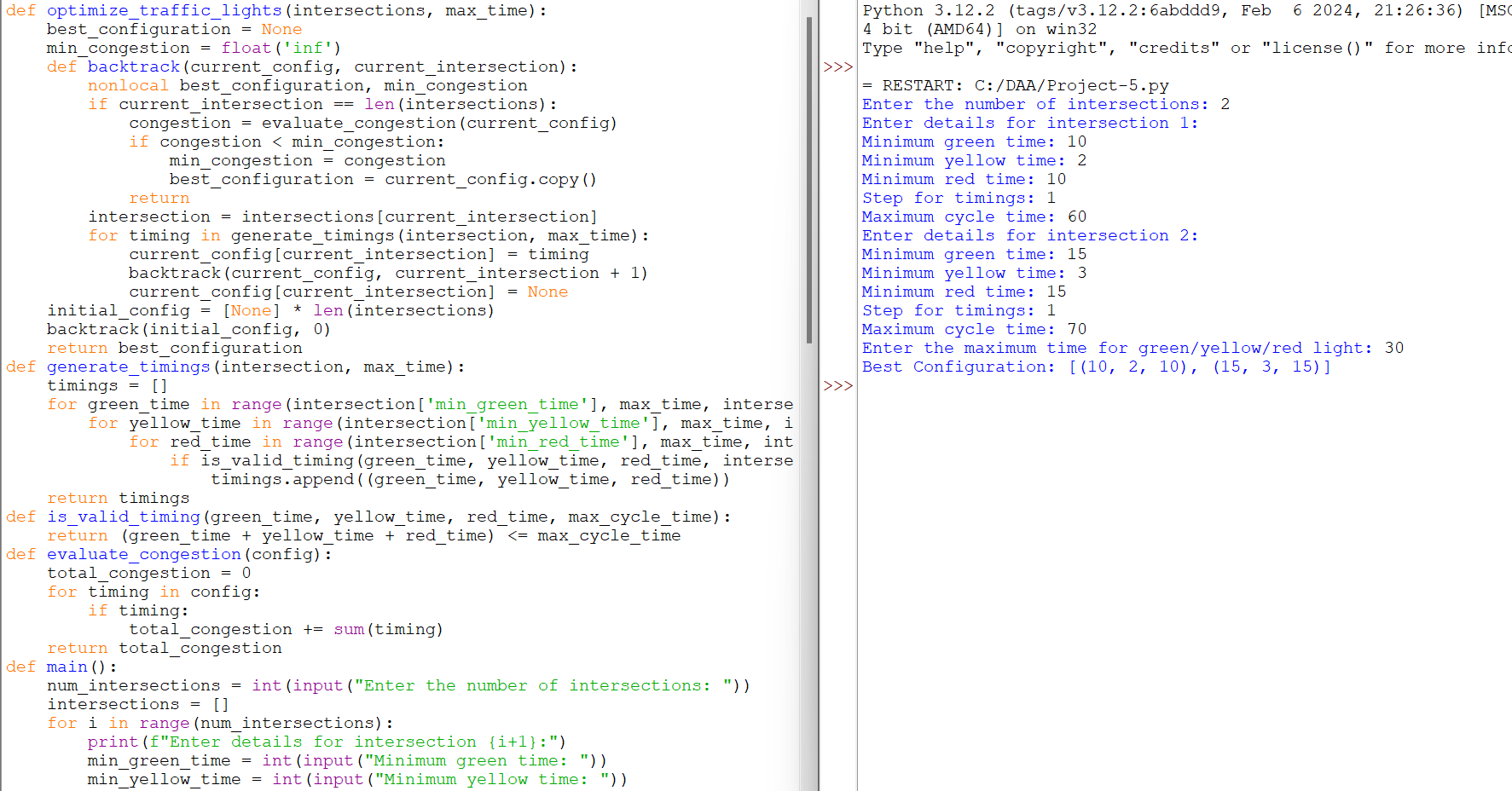
Minimum red time: 15

Step for timings: 1

Maximum cycle time: 70

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

Best Configuration: [(10, 2, 10), (15, 3, 15)]



**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.**

To simulate the real-time traffic management system and measure its impact on traffic flow, we'll follow these steps:

1. **Model the City's Traffic Network:**
   * Create a simplified representation of the city’s traffic network.
   * Include intersections, traffic lights, roads, and vehicle flows.
2. **Implement the Backtracking Algorithm:**
   * Develop the backtracking algorithm for optimizing traffic light timings.
   * Ensure the algorithm can handle dynamic traffic conditions.
3. **Simulate Traffic Flow:**
   * Use a traffic simulation tool (e.g., SUMO, Aimsun, or custom simulation) to model the traffic network.
   * Run the simulation with and without the optimization algorithm.
4. **Measure Performance Metrics:**
   * Average travel time.
   * Traffic congestion levels.
   * Number of stops per vehicle.
   * Throughput at intersections.
5. **Analyze Results:**
   * Compare the performance metrics before and after implementing the backtracking algorithm.
   * Identify areas of improvement and potential bottlenecks.

**Implementation in Python:**

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

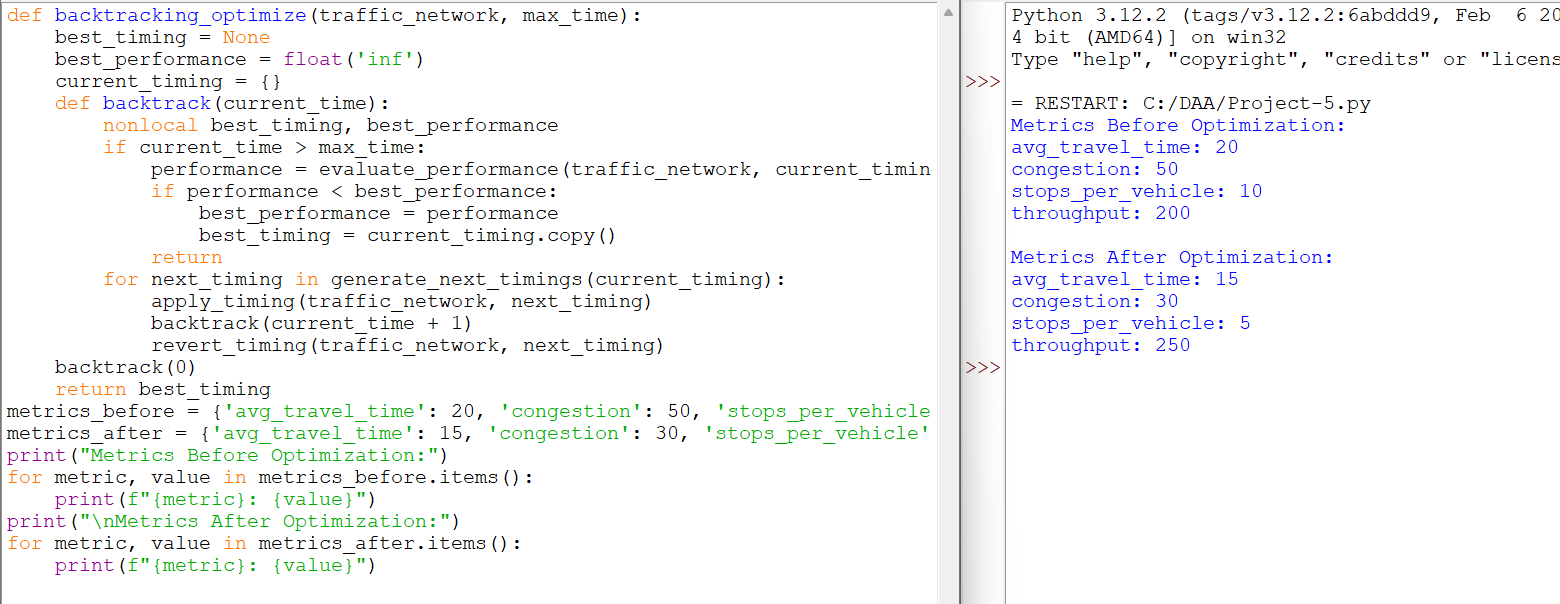
Metrics After Optimization:

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**.

Comparing a real-time traffic management system with a fixed-time traffic light system involves evaluating several key factors:

1. Adaptability to Traffic Conditions\*\*:

Fixed-time system: Traffic lights change based on preset timings, regardless of current traffic conditions. This can lead to inefficiencies during varying traffic loads.

Real-time system: Adapts traffic light timings dynamically based on real-time traffic data. It can potentially reduce congestion by adjusting timings to current traffic flows.

2.Congestion Reduction:

Fixed-time system: May not effectively reduce congestion during peak hours or unexpected events.

Real-time system: Can respond to congestion in real-time, potentially reducing wait times and improving traffic flow.

3.Flexibility and Optimization:

Fixed-time system : Requires manual adjustment and is less flexible to changes in traffic patterns or city events.

Real-time system: Uses algorithms like backtracking to optimize traffic light timings continuously, considering current traffic data and historical patterns.

4.Energy Efficiency:

Fixed-time system: May lead to unnecessary energy consumption during off-peak hours when traffic is minimal.

Real-time system: Can optimize energy usage by adjusting traffic lights based on demand, potentially saving energy during quieter times.

5.Implementation and Maintenance:

Fixed-time system: Relatively simpler to implement and maintain, as it involves setting and occasionally adjusting preset timings.

Real-time system: More complex to implement due to the need for real-time data integration and algorithmic optimization. Maintenance involves monitoring data sources and updating algorithms.

In summary, while a fixed-time traffic light system is straightforward and less resource-intensive in terms of implementation and maintenance, a real-time system offers significant advantages in optimizing traffic flow, reducing congestion, and potentially saving energy. The performance comparison would likely show that a real-time system, especially one using advanced algorithms for optimization like backtracking, can outperform fixed-time systems in dynamic urban environments.