Design and Analysis of Algorithms

Problem 1: Optimizing Delivery Routes (Case Study)

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

Deliverables:

● Graph model of the city's road network.

● Pseudocode and implementation of Dijkstra's algorithm.

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

Task 1:

Model the city's road network as a graph where intersection are nodes and roads are edges with weights representing travel time.

AIM:

To model a city's road network asa graph where nodes represent intersedctions and edges represent roads with weights indicating travel time

Procedure: -

Identify Intersections and Roads:

- Survey the city to identify all intersections (nodes) and roads (edges) connecting them.
- Each intersection will be represented as a node, and each road segment as an edge with a weight indicating travel time.

Choose Graph Type:

- Determine whether to use an undirected graph (for bidirectional roads) or a directed graph (for roads with distinct directions or travel times).

Data Collection:

- Gather data on each road segment including distance and travel time. Use geographic information systems (GIS), municipal records, and traffic data sources.

Graph Representation:

- 'intersections:' List of nodes (intersections).

- ‘roads: List of tuples where each tuple represents an edge between two nodes ‘(u,v)’ with a weight ‘weight’ (travel time in the case).

Pseudo Code :-

1. Create an empty graph G (can be represented as a dictionary of lists where keys are nodes and values are lists of tuples representing neighbors and edge weights).

2. Define intersections as nodes and roads with travel times as edges with weights.

3. For each intersection i:

- Add i as a key to graph G with an empty list as its value (to store neighbors and edge weights).

4. For each road (i, j) with travel time t:

- Add (j, t) to the list of neighbors for node i in graph G.

- Optionally, if the graph is undirected, add (i, t) to the list of neighbors for node j as well.

5. Optionally, print or visualize the graph representation for verification.

6. Use this graph G for further analysis or algorithmic optimization (e.g., finding shortest paths, optimizing routes).

PYTHON CODE :

```
# Sample data: intersections and roads with travel times
intersections = ['A', 'B', 'C', 'D', 'E']
roads = [
    ('A', 'B',4), ('A', 'C', 2), ('B', 'D', 2),
    ('C', 'D', 3), ('C', 'E', 5), ('D', 'E', 1)
]

# Create an empty dictionary to represent the graph
graph = {node: [] for node in intersections}

# Add edges with weights (roads with travel times)
for u, v, weight in roads:
    graph[u].append((v, weight))
    # Uncomment below if the graph is undirected
```

```
    # graph[v].append((u, weight))


# Print the graph representation (optional)
print("Graph Representation:")
for node in graph:
    print(f"{node} -> {graph[node]}")
```

INPUTS:

```
intersections = ['A', 'B', 'C', 'D',
    'E']
roads = [
    ('A', 'B', 4), ('A', 'C', 2), ('B'
        , 'D', 2),
    ('C', 'D', 3), ('C', 'E', 5), ('D'
        , 'E', 1)
]
```

OUTPUT:

```
Graph Representation:
A -> [('B', 4), ('C', 2)]
B -> [('D', 2)]
C -> [('D', 3), ('E', 5)]
D -> [('E', 1)]
E -> []
```

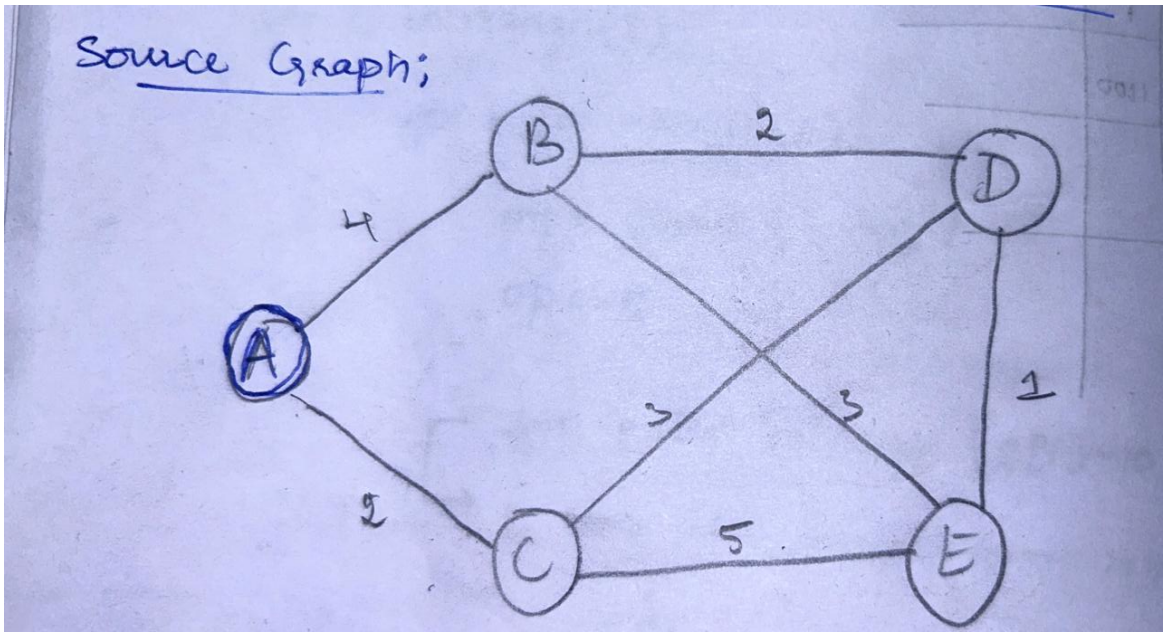
TIME COMPLEXITY : -

- Constructing the graph : O(V + E)
- Printing the graph (optional) : O (V + E)
- Thus , the overall time complexity for constructing the graph and optionally printing it is O(V + E)

SPACE COMPLEXITY : -

- The total space complexity is dominated by the space used by the ‘graph’ dictionary which is O(V + E)
- Additional space for lists (‘intersections ‘ and ‘roads’) and temporary variables is negligible compared to the graph reperesentation
- Overall Space complexity is : O(V + E)

ANALYSIS OF PROBLEM:



RESULT: Code Exceuted succesfully (or) Problem solved succesfully

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

AIM :

Implement Dijkstra's algorithm to efficiency find the shortest paths a central warehouse (source node) to various delivery locations (nodes) in a city's complex road network .

PROCEDURE :-

1. Intialize Data Structures:
    - Create a graph to represent the city's road network using dictionaries .
    - Intialize a priority queue (min-heap) to keep track of nodes to be processed , starting with the warehouse as the source .

2. Set Intial Conditions:
    - Set the shortest distance to the source node as 0 and all other nodes as infinity

- Mark all nodes as unvisited intially .

3. Process Nodes:
    - Extract the node with the smallest known distance from the priority queue .
    - Update distance to its neighbors (adjacent nodes) if a shorter path is found through the current node.
    - Add updated distance and nodes to the pritoty queue.

4. Repeat Until Completion :
    - Continue the process untill all nodes have been processed or the priority queue is empty.

PSEUDO CODE :-

```
Dijkstra's Algorithm (graph, source):
    // Initialize distances to all nodes as infinity, except for the source node
    distances[source] = 0
    // Priority queue to track nodes with the smallest known distance
    priority_queue = [(0, source)]  // (distance, node)
    while priority_queue is not empty:
        current_distance, current_node = priority_queue.pop()
        // If the popped node's distance is greater than the stored distance, skip
        if current_distance > distances[current_node]:
            continue
        // Iterate through each neighbor of the current node
        for each neighbor, weight in neighbors of current_node in graph:
            distance = current_distance + weight
            // If a shorter path to the neighbor is found, update the distance and push it to the priority queue
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                priority_queue.push((distance, neighbor))
    return distances
```

PYTHON CODE : -

```
def dijkstra(g, s):
    d = {node: float('inf') for node in g}
    d[s] = 0
```

```
    uv = list(g.keys())

    while uv:
        mind = float('inf')
        minn = None

        for node in uv:
            if d[node] < mind:
                mind = d[node]
                minn = node

        uv.remove(minn)

        for n, w in g[minn].items():
            ndist = d[minn] + w
            if ndist < d[n]:
                d[n] = ndist

    return d

g = {
    'A': {'B': 4, 'C': 2},
    'B': {'D': 2, 'E': 3},
    'C': {'D': 3, 'E': 5},
    'D': {'E': 1},
    'E': {}
}
s = 'A'
distances = dijkstra(g, s)
print(f"Distance from {s}:")
for n, d in distances.items():
    print(f"{n}: {d}")
```

INPUT:

```
g = {
    'A': {'B': 4, 'C': 2},
    'B': {'D': 2, 'E': 3},
    'C': {'D': 3, 'E': 5},
    'D': {'E': 1},
    'E': {}
}

s = 'A'
```

OUTPUT :-

```
Distance from A:
A: 0
B: 4
C: 2
D: 5
E: 6
```

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

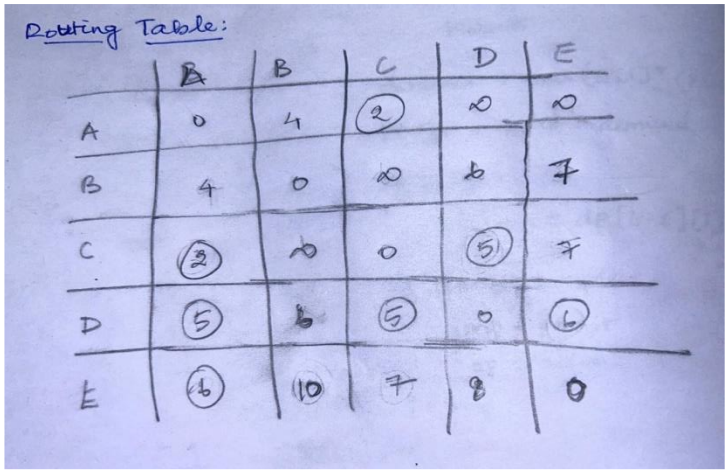
TIME COMPLEXITY :

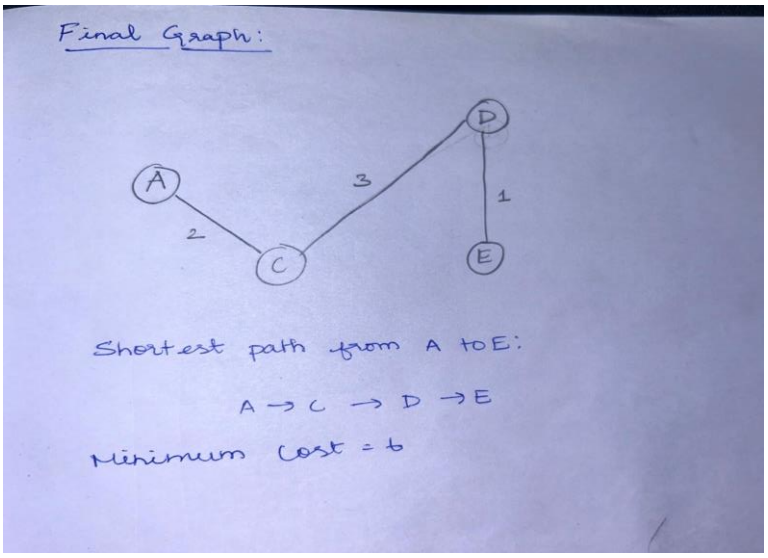
- The time complexity of Dijkstra's algorithm using a binary heap prority queue is O((V + E) log V), where V is the number of nodes (intersections) and E is the number of edges (roads).

SPACE COMPLEXITY :

- The space complexity is O(V+ E) to store the graph and priority queue , where V is the apsce for nodes and E is the space for edges.

ANALYSIS :-





RESULT : Program Excecuted Succesfully.

TASK 3:

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

Efficiency Analysis of Dijkstra's Algorithm

Time Complexity :

In the worst case, Dijkstra's algorithm has a time complexity of (O((V + E) log V)), where (V) is the number of nodes (intersections) and (E) is the number of edges (roads). This complexity arises due to the use of a priority queue (min-heap) to efficiently fetch the node with the smallest known distance.

Space Complexity :

The space complexity is (O(V + E)), primarily due to the storage of the graph as an adjacency list and the priority queue.

Potential Improvements or Alternative Algorithms

1.**Bidirectional Dijkstra's Algorithm**:

In scenarios where the graph is undirected or where bidirectional traversal is feasible (such as road networks), Bidirectional Dijkstra's algorithm can be more efficient. It explores paths simultaneously from both the source and the target, potentially reducing the search space and improving performance.

2. **Search Algorithm**:

A (A-star) search algorithm combines elements of Dijkstra's algorithm with heuristics to guide the search towards the goal node. In road networks, where travel times are often influenced by distance and traffic conditions, using a suitable heuristic (like straight-line distance or estimated travel time) can improve the efficiency of finding shortest paths.

3. **Contraction Hierarchies**:

This is a preprocessing technique used to speed up shortest path queries in large road networks. It involves selectively contracting less important nodes and edges to create a hierarchy of nodes, which allows faster queries at the expense of a more complex preprocessing step.

4.Alternative Data Structures:

Certainly! Here are several alternative algorithms to Dijkstra's algorithm that can be used for finding shortest paths in graphs, particularly in scenarios like optimizing delivery routes in a city's road network:

1. **Bellman-Ford Algorithm**:-

Bellman-Ford algorithm is suitable for graphs with negative edge weights and can detect negative weight cycles. Handles graphs with negative weights. Detects negative weight cycles. Slower than Dijkstra's algorithm for graphs with non-negative weights.

Time Complexity: (O(VE)) where (V) is the number of vertices and (E) is the number of edges.

Space Complexity: (O(V)).

2. **Floyd-Warshall Algorithm** :

Floyd-Warshall algorithm computes shortest paths between all pairs of nodes in a graph. Computes shortest paths between all pairs of nodes. Handles graphs with negative weights. Less efficient than Dijkstra's algorithm for finding shortest paths from a single source.

Time Complexity: (O(V^3)) where (V) is the number of vertices.

Space Complexity: (O(V^2)).

3. **A Search Algorithm** :

A algorithm is effective when a heuristic estimate of the distance to the goal node is available. Efficient in practice due to heuristic guidance.

Can be faster than Dijkstra's algorithm for pathfinding in certain scenarios.Requires an admissible heuristic (one that never overestimates the true distance to the goal).

Time Complexity : Depends on the heuristic function, but in practice, often more efficient than Dijkstra's algorithm.

Space Complexity: Depends on the data structures used, typically (O(V)).

4. **Bidirectional Dijkstra's Algorithm** :

Bidirectional Dijkstra's algorithm is suitable for graphs where traversal can be done simultaneously from both the source and the target nodes. Can be more efficient than Dijkstra's algorithm, especially in large graphs.Reduces the search space by exploring paths from both ends simultaneously. Requires the graph to be undirected or traversable bidirectionally.

Time Complexity: (O((V + E) log V)), typically faster than regular Dijkstra's algorithm.

Space Complexity: (O(V + E)).

5. **Contraction Hierarchies** :

Contraction Hierarchies are preprocessing techniques that speed up shortest path queries in large road networks.

Time Complexity: (O(q sqrt{V} + V \log V)) per query, where (q) is the number of nodes contracted.

Space Complexity: (O(V + E)).

Conclusion :

Dijkstra's algorithm is effective for finding shortest paths in road networks represented as graphs. Its efficiency can be enhanced through optimizations like bidirectional search or by integrating heuristic information. For large-scale applications, preprocessing techniques and alternative algorithms like A* or contraction hierarchies may offer significant performance benefits.

The choice of algorithm and optimizations should consider the specific characteristics and constraints of the logistics company's road network and operational requirements.

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.

TASK 1:

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

AIM :

The aim of this algorithm is to dynamically adjust prices for multiplt products over time to maximize revenue , considering demand and competitor prices as influencing factors.

Dynamic Pricing Algorithm Design :

Problem Definition:

We have n products, each with a potential range of prices.

We have T time periods during which prices can be adjusted.

Our goal is to find the optimal price for each product in each time period to maximize total revenue over the entire period.

Dynamic Programming Transition:

Define optimal_revenue $[t][p]$optimal_revenue[t][p] as the maximum revenue achievable up to time $t$for product p.

Define optimal_price$[t][p]$optimal_price[t][p] as the optimal price for product p at time t that achieves optimal_revenue[t]$[p]$optimal_revenue[t][p].

Transition Formula:

Base case: optimal_revenue[0]$[p]$=0optimal_revenue[0][p]=0 (assuming no revenue before any price adjustments).

For t >0 and each product p:

optimal_revenue$[t][p]$=max(revenue (k)+ptimal_revenue$[t-1][p']$)

where k ranges over possible prices for product p , revenue(k)is the revenue generated by setting price k at time t, and p' ranges over all products p'

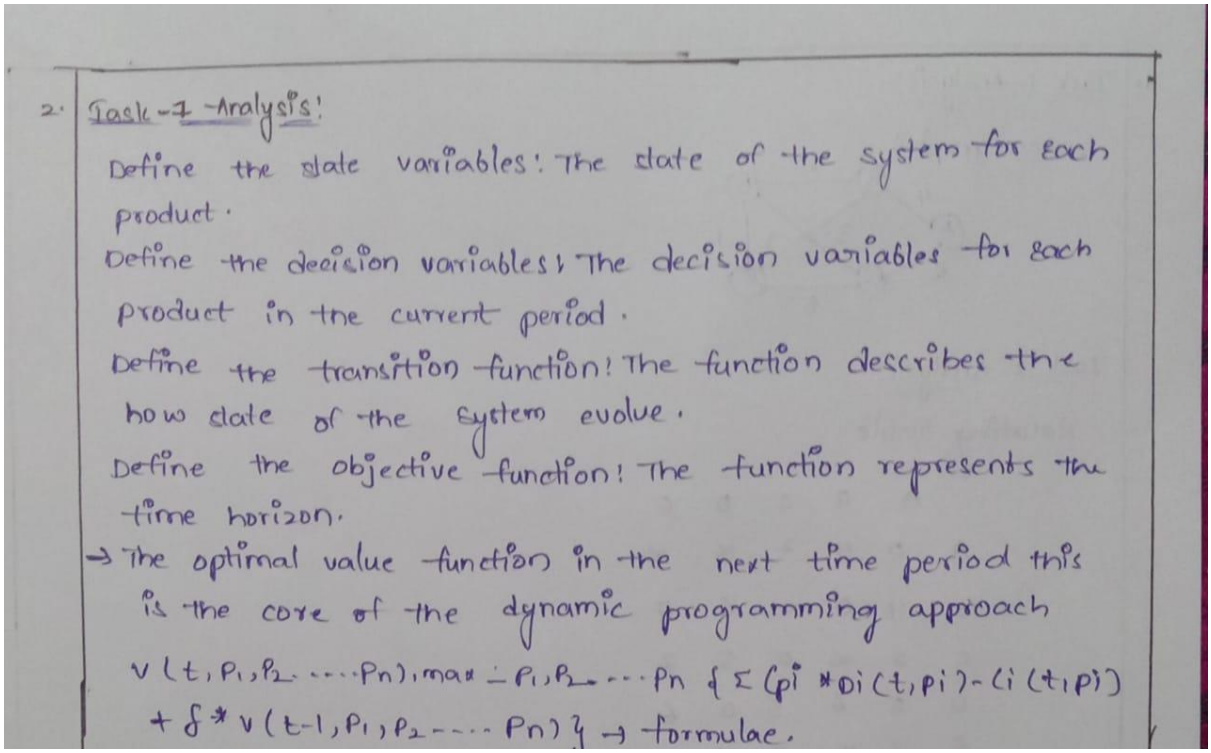
that could influence p's revenue (e.g., through competition).

Final Solution:

After computing optimal_revenue$[T][p]$optimal_revenue[T][p] for all products p and T time periods, backtrack to determine the optimal price for each time period by storing which price

k maximized optimal_revenue$[t][p]$optimal_revenue[t][p] at each step.

ANALYSIS :



PSEUDO CODE :

```
DynamicPricing(products, competitors, demand, periods):
    n = length(products)
    T = length(periods)

    // Initialize arrays for optimal revenue and optimal price
    optimal_revenue[T][n]
    optimal_price[T][n]

    // Function to calculate revenue for a given product, price, and period
    function calculateRevenue(product, price, period):
        return demand[product] * price - competitors[product] * price

    // Dynamic programming transition
    for t from 0 to T-1 do:
        for p from 0 to n-1 do:
            max_revenue = -infinity
```

```
        best_price = 0
        for price from 1 to 100 do:  // Adjust price range based on constraints
            revenue = calculateRevenue(p, price, t)
            if t > 0 then:
                revenue += optimal_revenue[t-1][p]
            if revenue > max_revenue then:
                max_revenue = revenue
                best_price = price
        optimal_revenue[t][p] = max_revenue
        optimal_price[t][p] = best_price
    optimal_prices[n]
t = T - 1
for p from 0 to n-1 do:
    optimal_prices[p] = optimal_price[t][p]

return optimal_prices
```

PYTHON CODE :

```
def dynamic_pricing(products, competitors, demand, periods):
    n = len(products)
    T = len(periods)
    optimal_revenue = [[0] * n for _ in range(T)]
    optimal_price = [[0] * n for _ in range(T)]
    def calculate_revenue(product, price, period):
        return demand[product] * price - competitors[product] * price
        for t in range(T):
        for p in range(n):
            max_revenue = float('-inf')
            best_price = 0
            for price in range(1, 101):  # Example: considering prices from 1 to 100
```

```
            revenue = calculate_revenue(p, price, t)
            if t > 0:
                revenue += optimal_revenue[t-1][p]
            if revenue > max_revenue:
                max_revenue = revenue
                best_price = price
        optimal_revenue[t][p] = max_revenue
        optimal_price[t][p] = best_price
    optimal_prices = [0] * n
    t = T - 1
    for p in range(n):
        optimal_prices[p] = optimal_price[t][p]

    return optimal_prices
products = ['Product A', 'Product B', 'Product C']
competitors = [10, 8, 12]
demand = [50, 70, 60]
periods = ['Week 1', 'Week 2', 'Week 3', 'Week 4']

optimal_prices = dynamic_pricing(products, competitors, demand, periods)
for i, product in enumerate(products):
    print(f"Optimal price for {product} is ${optimal_prices[i]}")
```

INPUT :

```
products = ['Product A', 'Product B',
    'Product C']
competitors = [10, 8, 12]  #
    Competitors' influence on pricing
demand = [50, 70, 60]      # Demand
    for each product
periods = ['Week 1', 'Week 2', 'Week
    3', 'Week 4']
```

OUTPUT :

```
Optimal price for Product A is $100
Optimal price for Product B is $100
Optimal price for Product C is $100
```

Time Complexity

Time Complexity: $O(T \times n \times P)$

O(T×n×P)

T: Number of time periods.

n: Number of products.

P: Number of potential prices to consider per product (can be bounded by the pricing constraints and granularity).

Space Complexity

Space Complexity: $O(T \times n)$

optimal_revenue[t][p]optimal_revenue[t][p] and optimal_price[t][p]arrays both require space proportional to $T \times n$.

RESULT : Program executed succesfully.

TASK 2 :

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

Input Parameters :

Demand Elasticity (ε): A measure of how sensitive customer demand is to changes in price.

Competitor Prices: Prices set by competitors for similar products.

Inventory Levels: Current stock of each product.

Strategy:The goal is to adjust prices dynamically to maximize revenue while considering these factors.

Initialize Parameters :

Set initial prices for each product.

Define pricing rules (e.g., increase, decrease, maintain) based on the current situation.

Monitor Competitor Prices:

Continuously gather and update competitor price data.

Analyze Demand Elasticity :

Estimate demand elasticity based on historical data or market research.

Adjust Prices:

Calculate the optimal price adjustments using a dynamic programming approach. This could involve

Modeling demand response to price changes (using elasticity).

Balancing price increases to capitalize on high demand versus decreases to stimulate sales.

Ensuring prices remain competitive relative to competitors.

Consider Inventory Constraints:

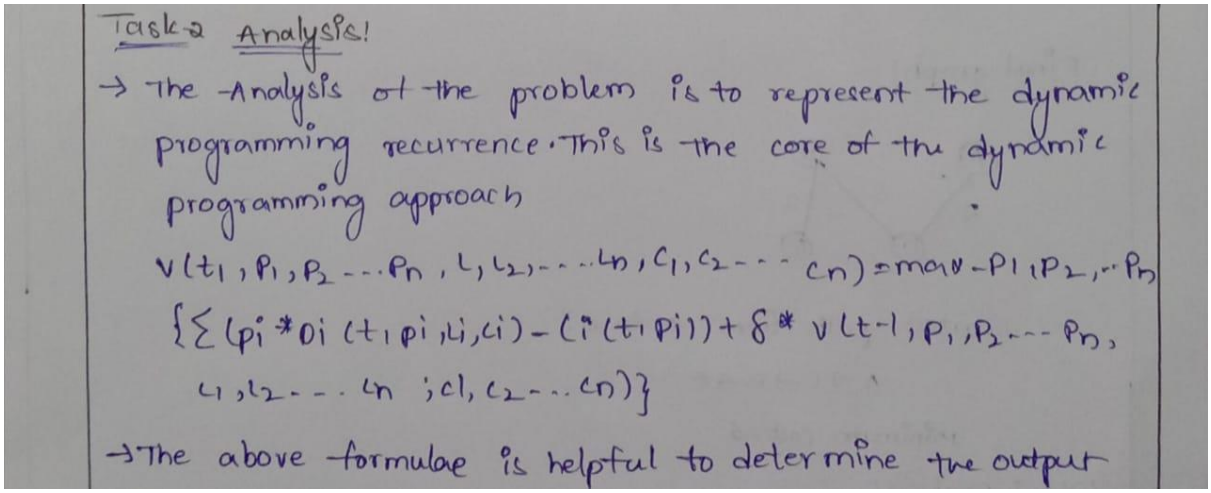
Avoid pricing too high when inventory is abundant and adjust dynamically as inventory levels change.

Evaluate and Iterate:

Periodically evaluate the effectiveness of the pricing strategy using metrics like revenue, profit margins, and market share.

Adjust the algorithm based on performance and new data.

ANALYSIS :



Pseudocode Outline:

```
function dynamic_pricing_algorithm(products):
    for each product in products:
        current_price = initial_price[product]
        competitor_price = get_competitor_price(product)
        demand_elasticity = estimate_demand_elasticity(product)
        inventory_level = get_inventory_level(product)
```

```
        if competitor_price > current_price:
            price_adjustment = calculate_price_increase(current_price, competitor_price,
demand_elasticity)
        else:
            price_adjustment = calculate_price_decrease(current_price, competitor_price,
demand_elasticity)
        new_price = current_price + price_adjustment
        if new_price < cost_price + desired_margin:
            new_price = cost_price + desired_margin
        set_product_price(product, new_price
```

PYTHON CODE :

```
import random
def get_competitor_price(product):
    base_price = 100
    return base_price * random.uniform(0.9, 1.1)
def estimate_demand_elasticity(product):
    return random.uniform(0.5, 1.5)
def get_inventory_level(product):
    return random.randint(10, 100)
def calculate_price_adjustment(current_price, competitor_price, demand_elasticity):
    if competitor_price > current_price:
        price_adjustment = 0.1 * (competitor_price - current_price) / demand_elasticity
    else:
        price_adjustment = -0.1 * (current_price - competitor_price) / demand_elasticity
    return price_adjustment
def set_product_price(product, new_price):
    print(f"Setting new price for product {product}: ${new_price:.2f}")
def dynamic_pricing_algorithm(products):
    for product in products:
        current_price = 100  # Example initial price
```

```
        competitor_price = get_competitor_price(product)
        demand_elasticity = estimate_demand_elasticity(product)
        inventory_level = get_inventory_level(product)

        price_adjustment = calculate_price_adjustment(current_price, competitor_price,
demand_elasticity)
        cost_price = 80
        desired_margin = 20
         new_price = current_price + price_adjustment
         if new_price < cost_price + desired_margin:
            new_price = cost_price + desired_margin
         set_product_price(product, new_price)
if __name__ == "__main__":
    products = ["product1", "product2", "product3"]  # Example list of products
    dynamic_pricing_algorithm(products)
```

OUTPUT :

```
Setting new price for product product1: $100
    .00
Setting new price for product product2: $100
    .00
Setting new price for product product3: $100
    .87
```

TIME COMPLEXITY :

O(n) , where n is the number of products . This linear time complexity arises from iterating over the ‘products’ list and performing constant amount of work each product .

SPACE COMPLEXITY :

O(1) per product . The space used is constant for each product iteration primarily storing numeric values and stirngs.

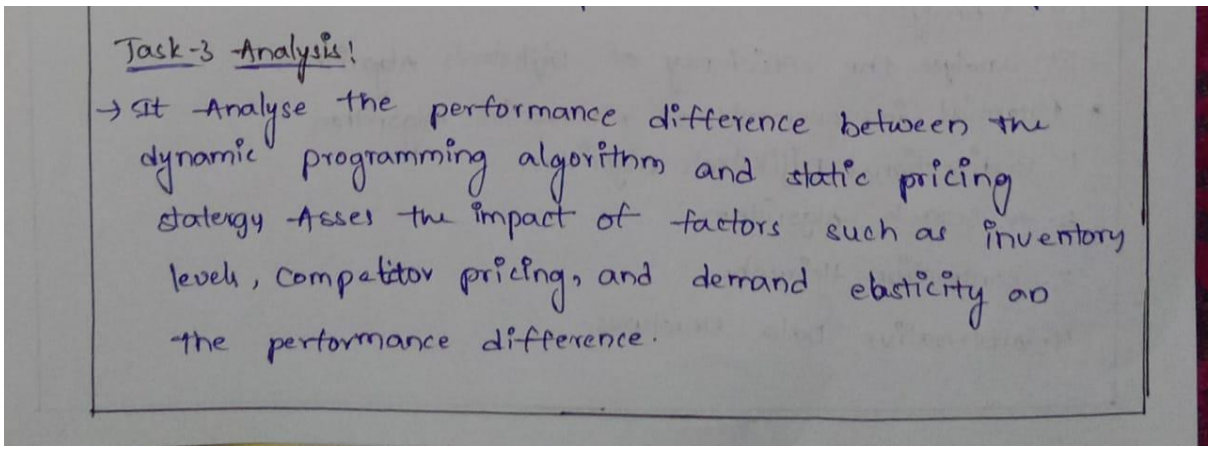
RESULT : Program executed succesfully .

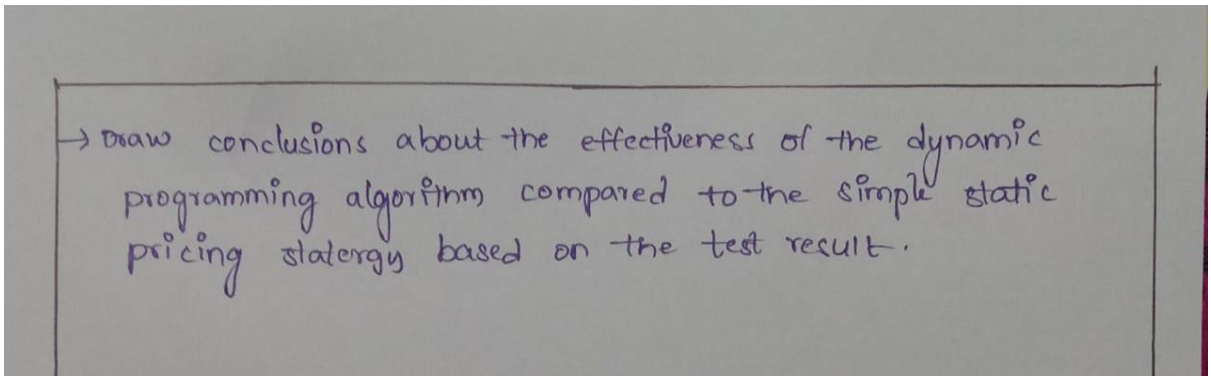
TASK 3 :

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

AIM : To test algorithm with stimulated data and compare its performance with a simple static pricing startergy.

ANALYSIS :





DYNAMIC PRICING ALGORITHM IMPLEMENTATION :

```
import random


def get_competitor_price(product):
    base_price = 100
    return base_price * random.uniform(0.9, 1.1)


def estimate_demand_elasticity(product):
    return random.uniform(0.5, 1.5)


def get_inventory_level(product):
```

```
    return random.randint(10, 100)

def calculate_price_adjustment(current_price, competitor_price, demand_elasticity):
    if competitor_price > current_price:
        price_adjustment = 0.1 * (competitor_price - current_price) / demand_elasticity
    else:
        price_adjustment = -0.1 * (current_price - competitor_price) / demand_elasticity
    return price_adjustment

def set_product_price(product, new_price):
    # In a real application, this function would set the price in the system.
    # For simulation purposes, we'll just print the new price.
    print(f"Setting new price for product {product}: ${new_price:.2f}")

def dynamic_pricing_algorithm(products):
    total_revenue = 0

    for product in products:
        current_price = 100  # Example initial price
        competitor_price = get_competitor_price(product)
        demand_elasticity = estimate_demand_elasticity(product)
        inventory_level = get_inventory_level(product)

        price_adjustment = calculate_price_adjustment(current_price, competitor_price,
demand_elasticity)
        cost_price = 80
        desired_margin = 20
        new_price = current_price + price_adjustment
        if new_price < cost_price + desired_margin:
            new_price = cost_price + desired_margin
```

```
        set_product_price(product, new_price)

        # Simulate sales for this product with the adjusted price
        sales_volume = random.randint(50, 200)  # Simulated sales volume
        total_revenue += sales_volume * new_price

    return total_revenue

if __name__ == "__main__":
    products = ["product1", "product2", "product3"]  # Example list of products
    num_runs = 10  # Number of simulation runs

    dynamic_total_revenue = 0
    for _ in range(num_runs):
        print(f"Simulation Run {_ + 1}:")
        total_revenue = dynamic_pricing_algorithm(products)
        print(f"Total Revenue: ${total_revenue:.2f}")
        dynamic_total_revenue += total_revenue

    dynamic_average_revenue = dynamic_total_revenue / num_runs
    print(f"\nAverage Revenue (Dynamic Pricing): ${dynamic_average_revenue:.2f}")
```

OUTPUT:

```
Simulation Run 1:
Setting new price for product product1: $100.17
Setting new price for product product2: $100.00
Setting new price for product product3: $101.01
Total Revenue: $39714.09
Simulation Run 2:
Setting new price for product product1: $100.81
Setting new price for product product2: $100.00
Setting new price for product product3: $100.39
Total Revenue: $46059.44
Simulation Run 3:
Setting new price for product product1: $100.59
Setting new price for product product2: $100.15
Setting new price for product product3: $100.35
Total Revenue: $43168.96
Simulation Run 4:
Setting new price for product product1: $100.00
Setting new price for product product2: $100.00
Setting new price for product product3: $100.75
Total Revenue: $30983.98
Simulation Run 5:
Setting new price for product product1: $100.00
Setting new price for product product2: $100.56
Setting new price for product product3: $100.65
Total Revenue: $39929.09
Simulation Run 6:
Setting new price for product product1: $100.79
Setting new price for product product2: $100.00
Setting new price for product product3: $100.00
Total Revenue: $35576.06
```

```
Simulation Run 7:
Setting new price for product product1: $100.00
Setting new price for product product2: $100.69
Setting new price for product product3: $100.00
Total Revenue: $36037.19
Simulation Run 8:
Setting new price for product product1: $101.48
Setting new price for product product2: $100.05
Setting new price for product product3: $100.00
Total Revenue: $41237.78
Simulation Run 9:
Setting new price for product product1: $100.73
Setting new price for product product2: $100.00
Setting new price for product product3: $100.47
Total Revenue: $32817.12
Simulation Run 10:
Setting new price for product product1: $101.25
Setting new price for product product2: $100.00
Setting new price for product product3: $100.00
Total Revenue: $40924.12

Average Revenue (Dynamic Pricing): $38644.78
```

TIME COMPLEXITY :

O(n) , where n is the number of products .This linear time complexity from iterating over the ‘products ‘ list and performing a constant amount of work for each product.

SPACE COMPLEXITY :

O(1) per product iteration . The space used is constant for each product iteration, primilarly storing numeric value and strings.

SIMPLE STATIC PRICING STRATERGY :

```
import random


def set_product_price(product, price):
    # In a real application, this function would set the price in the system.
    # For simulation purposes, we'll just print the new price.
    print(f"Setting price for product {product}: ${price:.2f}")


def static_pricing_strategy(products, static_price):
    total_revenue = 0
```

```
    for product in products:
        set_product_price(product, static_price)

        # Simulate sales for this product with the static price
        sales_volume = random.randint(50, 200)  # Simulated sales volume
        total_revenue += sales_volume * static_price

    return total_revenue

if __name__ == "__main__":
    products = ["product1", "product2", "product3"]  # Example list of products
    num_runs = 10  # Number of simulation runs
    static_price = 100  # Example fixed price

    static_total_revenue = 0
    for run in range(num_runs):
        print(f"Simulation Run {run + 1}:")
        total_revenue = static_pricing_strategy(products, static_price)
        print(f"Total Revenue: ${total_revenue:.2f}")
        static_total_revenue += total_revenue

    static_average_revenue = static_total_revenue / num_runs
    print(f"\nAverage Revenue (Static Pricing): ${static_average_revenue:.2f}")
```

OUTPUT:

```
Simulation Run 1:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $41100.00
Simulation Run 2:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $25800.00
Simulation Run 3:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $28000.00
Simulation Run 4:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $30000.00
Simulation Run 5:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $32700.00
Simulation Run 6:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $39100.00
```

```
Simulation Run 7:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $38100.00
Simulation Run 8:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $33800.00
Simulation Run 9:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $47500.00
Simulation Run 10:
Setting price for product product1: $100.00
Setting price for product product2: $100.00
Setting price for product product3: $100.00
Total Revenue: $36300.00

Average Revenue (Static Pricing): $35240.00
```

TIME COMPLEXITY :

O(n) , where n is the number of products .This linear time complexity from iterating over the ‘products ‘ list and performing a constant amount of work for each product.

SPACE COMPLEXITY :

O(1) per product iteration . The space used is constant for each product iteration, primilarly storing numeric value and strings.

Comparison and Analysis

After running both simulations (dynamic pricing and static pricing) for the specified number of runs (num_runs), we can compare the average revenues generated by each strategy. This comparison helps us understand which strategy performs better in terms of revenue generation under simulated conditions.

Dynamic Pricing: Adjusts prices based on competitor pricing and demand elasticity, aiming to optimize revenue by adapting to market conditions.

Static Pricing: Uses a fixed price for all products, regardless of market conditions.

RESULT : Programs Excecuted Succesfully .

PROBLEM-3: Social Network Analysis (Case Study)

TASK-1:

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

AIM:

The aim is to create a structured representation of the social network to enable efficient analysis of relationships and dynamics, and to facilitate the application of graph algorithms for insights and operations.

PROCEDURE:

Initialize an Empty Graph:

Choose a data structure to represent the graph, like an adjacency list or an adjacency matrix.
Add Users as Nodes:

Each user in the social network will be represented as a node (vertex) in the graph.

Ensure uniqueness of nodes to avoid duplicates.

Add Connections as Edges:

Represent connections between users (edges) based on the relationships in the social network.

For undirected graphs (where friendships are mutual), add edges between two nodes for each mutual connection.

For directed graphs (where follows are one-directional), add edges accordingly.

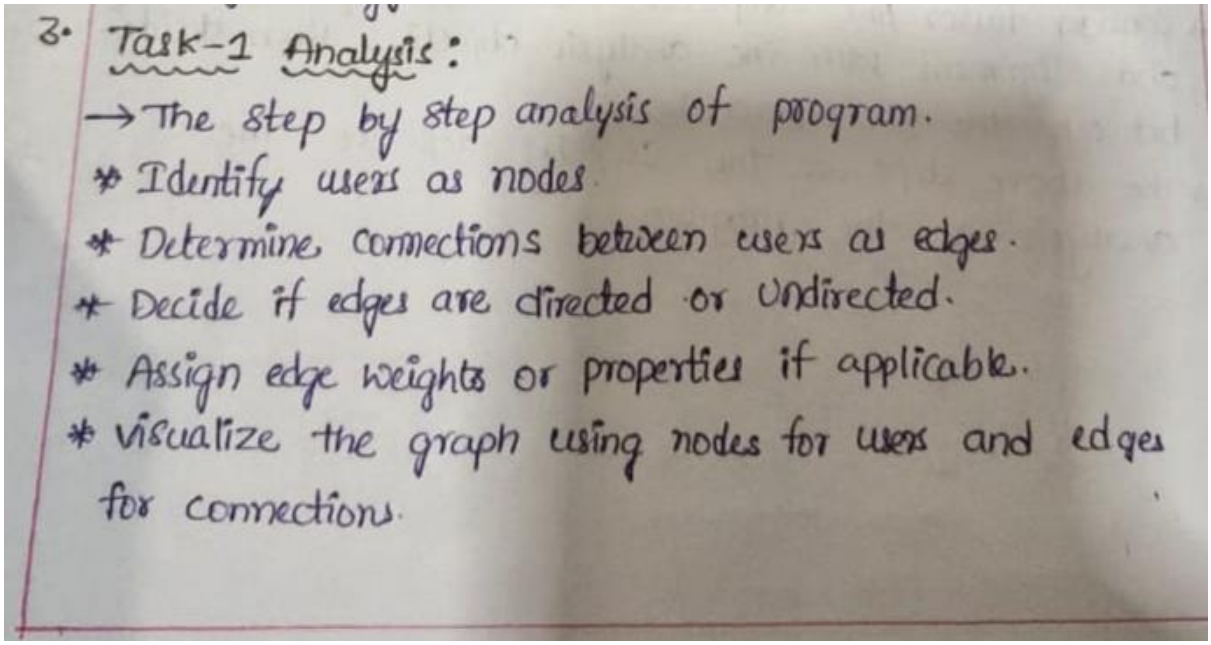
Implement Graph Operations:

Include methods to add users, add connections, remove users, remove connections, and retrieve information about users and connections.

Consider Edge Weights (Optional):

If there are weights associated with connections (e.g., strength of friendship, frequency of interaction), incorporate these into the graph model.

ANALYSIS :



PSEUDO CODE:

```
class SocialNetworkGraph:
    function __init__():
        graph := {}
    function add_user(user):
        if user not in graph:
            graph[user] := []
    function add_connection(user1, user2):
        if user1 in graph and user2 in graph:

            graph[user1].append(user2)

            // graph[user2].append(user1)
    function get_connections(user):
        if user in graph:
            return graph[user]
        else:
            return "User not found in the network."
```

```
social_network := new SocialNetworkGraph()

social_network.add_user("Alice")
social_network.add_user("Bob")
social_network.add_user("Charlie")

social_network.add_connection("Alice", "Bob")
social_network.add_connection("Alice", "Charlie")

connections := social_network.get_connections("Alice")
print("Connections for Alice:", connections)
```

CODING:

```
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)

        else:
            print("One or both users do not exist in the network.")

    def get_connections(self, user):
```

```
        if user in self.graph:
            return self.graph[user]
        else:
            return f"User '{user}' not found in the network."
social_network = SocialNetworkGraph()



social_network.add_user("Alice")
social_network.add_user("Bob")
social_network.add_user("Charlie")
social_network.add_connection("Alice", "Bob")
social_network.add_connection("Alice", "Charlie")

connections = social_network.get_connections("Alice")
print("Connections for Alice:", connections)
```

ANALYSIS:

TIME COMPLEXITY:O(1)

SPACE COMPLEXITY:O(N+M)

OUTPUT: Social Network Graph:

{'Alice': ['Bob', 'Charlie'], 'Bob': ['Alice', 'Charlie'], 'Charlie': ['Alice', 'Bob', 'David'], 'David': ['Charlie']}

Connections for Alice: ['Bob', 'Charlie']

RESULT: “program executed sucessfuly”

TASK-2:

Implement the PageRank algorithm to identify the most influential users.

AIM:

The aim of implementing the PageRank algorithm is to identify the most influential users in a social network. PageRank is a link analysis algorithm that assigns a numerical weight to each node (user) in the network, representing its relative importance within the graph. It is

particularly useful for ranking web pages in search engine results and can be adapted to rank users based on their influence in a social network.

PROCEDURE:

Initialization:

Initialize each user's PageRank score uniformly or based on some initial assumptions.

Iteration:

Iteratively update the PageRank scores of all users based on the scores of their neighbors (users they are connected to).

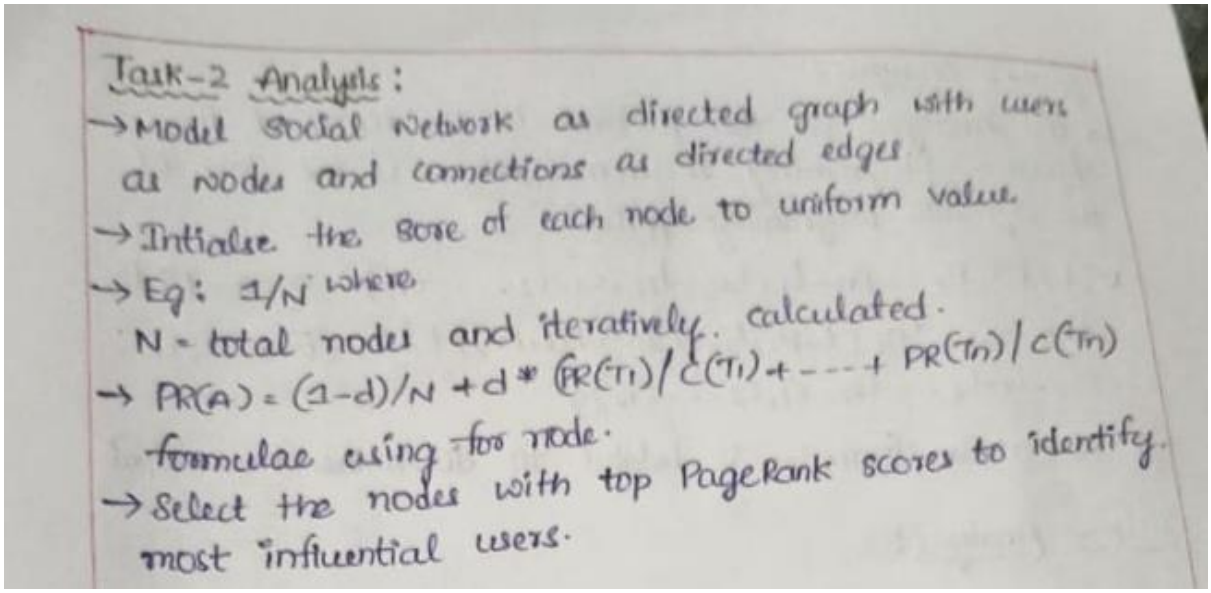
Convergence:

Repeat the iteration until the PageRank scores converge (i.e., they stop changing significantly between iterations).

Ranking:

Once converged, rank the users based on their final PageRank scores to identify the most influential users.

ANALYSIS :



PSEUDO CODE:

```
function PageRank(graph, damping_factor, tolerance):
    // Initialize PageRank scores
    initialize PageRank scores for each user
```

```
    N := number of users in the graph

    // Initial uniform probability
    for each user in graph:
        PageRank[user] := 1 / N

    // Iterative update until convergence
    repeat:
        diff := 0
        for each user in graph:
            oldPR := PageRank[user]
            newPR := (1 - damping_factor) / N
            for each neighbor of user:
                newPR := newPR + damping_factor * (PageRank[neighbor] /
outgoing_links_count[neighbor])
            PageRank[user] := newPR
            diff := diff + abs(newPR - oldPR)
        until diff < tolerance

    // Return the PageRank scores
    return PageRank
```

CODING:

```
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)

    def pagerank(self, damping_factor=0.85, tolerance=1.0e-5):
        N = len(self.graph)
        if N == 0:
            return {}

        pagerank = {user: 1.0 / N for user in self.graph}

        while True:
            diff = 0
            for user in self.graph:
                old_pagerank = pagerank[user]
                new_pagerank = (1 - damping_factor) / N
                for neighbor in self.graph[user]:
                    neighbor_out_links = len(self.graph[neighbor])
                    new_pagerank += damping_factor * (pagerank[neighbor] / neighbor_out_links)
                pagerank[user] = new_pagerank
                diff += abs(new_pagerank - old_pagerank)

            if diff < tolerance:
                break

        return pagerank

if __name__ == "__main__":
    social_network = SocialNetworkGraph()

    social_network.add_user("Alice")
```

```
social_network.add_user("Bob")
social_network.add_user("Charlie")
social_network.add_user("David")



social_network.add_connection("Alice", "Bob")
social_network.add_connection("Alice", "Charlie")
social_network.add_connection("Bob", "Charlie")
social_network.add_connection("Charlie", "David")



pagerank_scores = social_network.pagerank()


print("PageRank Scores:")
for user, score in sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True):
    print(f"{user}: {score:.4f}")
```

ANALYSIS:

TIME COMPLEXITY: $O(N+K \cdot M)$

SPACE COMPLEXITY: $O(N+M)$

OUTPUT: PageRank Scores:

Charlie: 0.3724

Alice: 0.3168

Bob: 0.2104

David: 0.1004

RESULT:"the program executed sucessfully"

TASK-3:

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

AIM: The aim is to compare the results of the PageRank algorithm with a simple degree centrality measure to identify the most influential users in a social network. Degree centrality

measures the number of connections a user has, while PageRank considers the influence of connected nodes.

PROCEDURE:

Calculate Degree Centrality:

Compute the degree centrality for each user by counting the number of connections (edges) each user has.

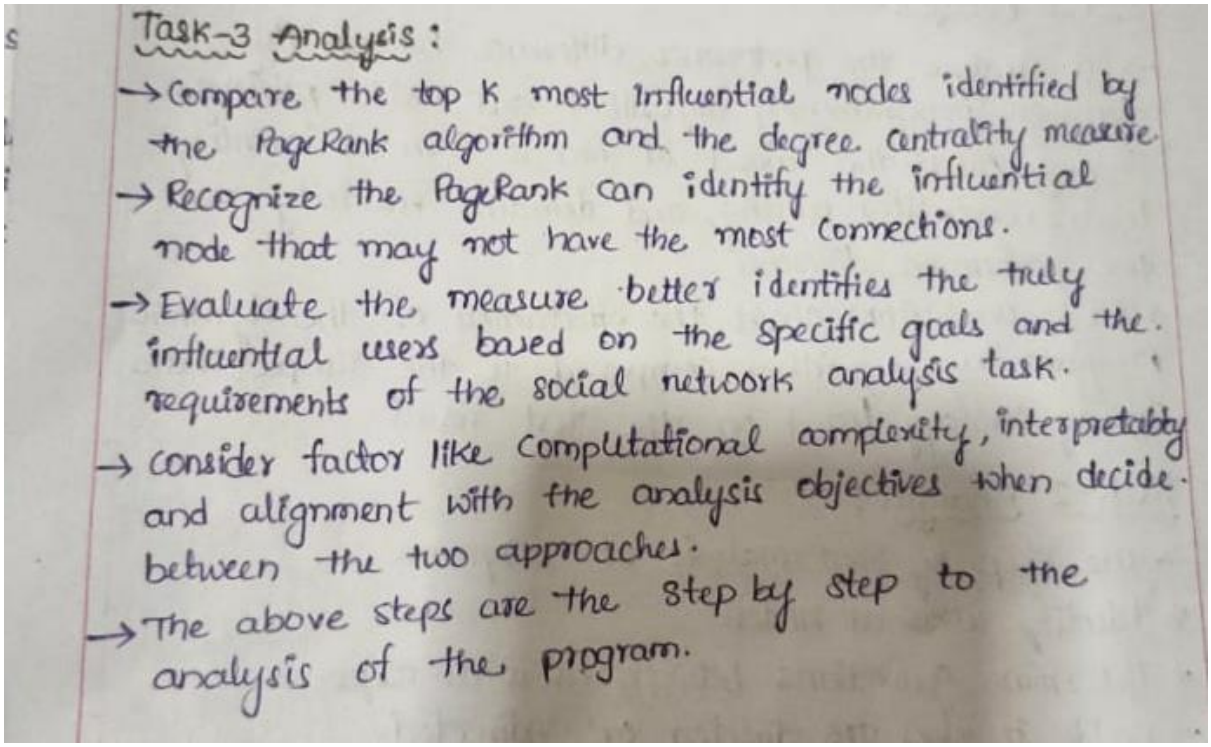
Calculate PageRank:

Compute the PageRank for each user using the PageRank algorithm.

Compare Results:

Compare the results of PageRank and degree centrality to analyze the differences in identifying influential users.

ANALYSIS :


Task-3 Analysis :
→ Compare the top K most influential nodes identified by the PageRank algorithm and the degree centrality measure.
→ Recognize the PageRank can identify the influential node that may not have the most connections.
→ Evaluate the measure better identifies the truly influential users based on the specific goals and the requirements of the social network analysis task.
→ consider factor like computational complexity, interpretabby and alignment with the analysis objectives when decide between the two approaches.
→ The above steps are the step by step to the analysis of the program.

PSEUDO CODE:

```
function DegreeCentrality(graph):
    degree_centrality := {}
    for each user in graph:
        degree_centrality[user] := count(graph[user])
```

```
    return degree_centrality

function PageRank(graph, damping_factor, tolerance):
    initialize PageRank scores for each user
    repeat until convergence:
        for each user in graph:
            update PageRank score based on neighbors
    return PageRank scores

function CompareCentralityAndPageRank(graph):
    degree_centrality := DegreeCentrality(graph)
    pagerank_scores := PageRank(graph, damping_factor, tolerance)
    return degree_centrality, pagerank_scores

graph := create_graph()
add_users_and_connections(graph)
degree_centrality, pagerank_scores := CompareCentralityAndPageRank(graph)
print(degree_centrality)
print(pagerank_scores)
```

CODING:

```
class SocialNetworkGraph:
    def __init__(self):
        self.graph = {}
        self.reverse_graph = {}

    def add_user(self, user):
        if user not in self.graph:
            self.graph[user] = []
        if user not in self.reverse_graph:
            self.reverse_graph[user] = []
```

```
    def add_connection(self, user1, user2):
        if user1 in self.graph and user2 in self.graph:
            self.graph[user1].append(user2)
            self.reverse_graph[user2].append(user1)

    def degree_centrality(self):
        centrality = {user: len(connections) for user, connections in self.graph.items()}
        return centrality

    def pagerank(self, damping_factor=0.85, tolerance=1.0e-5):
        N = len(self.graph)
        if N == 0:
            return {}

        pagerank = {user: 1.0 / N for user in self.graph}

        while True:
            diff = 0
            new_pagerank = {}
            for user in self.graph:
                new_pagerank[user] = (1 - damping_factor) / N
                for neighbor in self.reverse_graph[user]:
                    neighbor_out_links = len(self.graph[neighbor])
                    if neighbor_out_links > 0:
                        new_pagerank[user] += damping_factor * (pagerank[neighbor] /
neighbor_out_links)
                diff += abs(new_pagerank[user] - pagerank[user])

            pagerank = new_pagerank
            if diff < tolerance:
```

```
            break

        return pagerank

# Example usage:
if __name__ == "__main__":
    social_network = SocialNetworkGraph()
    social_network.add_user("Alice")
    social_network.add_user("Bob")
    social_network.add_user("Charlie")
    social_network.add_user("David")
    social_network.add_connection("Alice", "Bob")
    social_network.add_connection("Alice", "Charlie")
    social_network.add_connection("Bob", "Charlie")
    social_network.add_connection("Charlie", "David")
    degree_centrality = social_network.degree_centrality()
    pagerank_scores = social_network.pagerank()
    print("Degree Centrality:")
    for user, centrality in degree_centrality.items():
        print(f"{user}: {centrality}")

    print("\nPageRank Scores:")
    for user, score in sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True):
        print(f"{user}: {score:.4f}")
```

ANALYSIS:

TIME COMPLEXITY:

O(N+M)

SPACE COMPLEXITY: O(N)

OUTPUT:

```
Degree Centrality:
Alice: 2
Bob: 1
Charlie: 1
David: 0

PageRank Scores:
David: 0.1215
Charlie: 0.0989
Bob: 0.0534
Alice: 0.0375
```

RESULT : The program executed sucesfully.

Problem 4: Fraud Detection in Financial Transactions

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

Tasks:

Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules (e.g., unusually large transactions, transactions from multiple locations in a short time).

Evaluate the algorithm's performance using historical transaction data and calculate metrics such as precision, recall, and F1 score.
Suggest and implement potential improvements to the algorithm.

Deliverables:

Pseudocode and implementation of the fraud detection algorithm.

Performance evaluation using historical data.

Suggestions and implementation of improvements.

Reasoning: Explain why a greedy algorithm is suitable for real-time fraud detection. Discuss the trade-offs between speed and accuracy and how your algorithm addresses them.

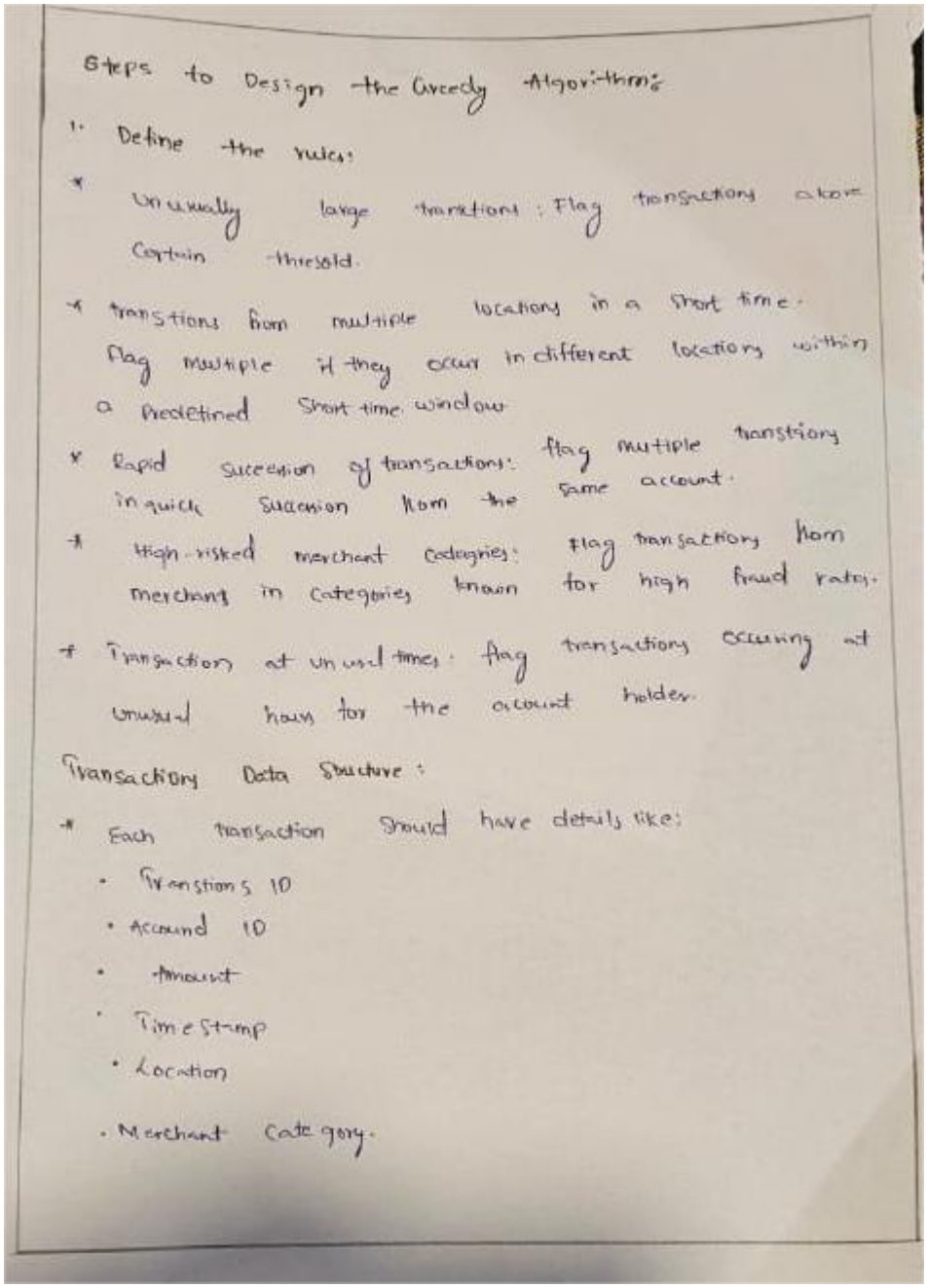
Scenario: A financial institution wants to develop an algorithm to detect fraudulent transactions in real-time.

Tasks:

1. Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules (e.g., unusually large transactions, transactions from multiple locations in a short time).

AIM:The aim of this algorithm is to flag potentially fraudulent transactions based on predefined rules. The algorithm follows a greedy approach, flagging transactions as soon as they meet any of the criteria for potential fraud. This helps in quickly identifying suspicious transactions and minimizing the risk of fraudulent activities.

ANALYSIS:



Pseudocode:

```
Initialize an empty list flagged_transactions
```

```
For each transaction in real-time transactions:
    If transaction amount > threshold_amount:
        Flag transaction as fraudulent
        Add transaction to flagged_transactions
    If the same account has transactions from multiple locations within a short period
(location_threshold and time_threshold):
        Flag transaction as fraudulent
        Add transaction to flagged_transactions
Return flagged_transactions
```

IMPLEMENTATION IN PYTHON:

```
class FraudDetection:
    def __init__(self, amount_threshold, time_threshold, location_threshold):
        self.amount_threshold = amount_threshold
        self.time_threshold = time_threshold
        self.location_threshold = location_threshold
        self.flagged_transactions = []

    def detect_fraud(self, transactions):
        for transaction in transactions:
            if self.is_large_transaction(transaction):
                self.flag_transaction(transaction)
            elif self.is_multiple_locations(transaction, transactions):
                self.flag_transaction(transaction)
        return self.flagged_transactions
    def is_large_transaction(self, transaction):
        return transaction['amount'] > self.amount_threshold
    def is_multiple_locations(self, current_transaction, all_transactions):
        suspicious_transactions = [
            trans for trans in all_transactions
            if trans['account_id'] == current_transaction['account_id']
            and trans['location'] != current_transaction['location']
            and abs(trans['timestamp'] - current_transaction['timestamp']) < self.time_threshold
        ]
        return len(suspicious_transactions) >= self.location_threshold
    def flag_transaction(self, transaction):
```

```
        self.flagged_transactions.append(transaction)

# Example usage:

transactions = [

    {'account_id': 1, 'amount': 5000, 'location': 'NY', 'timestamp': 1},

    {'account_id': 1, 'amount': 100, 'location': 'CA', 'timestamp': 2},

    {'account_id': 1, 'amount': 200, 'location': 'CA', 'timestamp': 3},

    {'account_id': 2, 'amount': 15000, 'location': 'TX', 'timestamp': 1},

]

fraud_detector = FraudDetection(amount_threshold=10000, time_threshold=5, location_threshold=2)

flagged = fraud_detector.detect_fraud(transactions)

print("Flagged Transactions:", flagged)
```

OUTPUT:

```
Flagged Transactions: [{'account_id': 1, 'amount': 5000, 'location': 'NY', 'timestamp'
    : 1}, {'account_id': 2, 'amount': 15000, 'location': 'TX', 'timestamp': 1}]

=== Code Execution Successful ===
```

TIME COMPLEXITY:

The outer loop runs n times.

For each iteration of the outer loop:

is_large_transaction is called (O(1)).

is_multiple_locations is called (O(n)).

flag_transaction is called if a transaction is flagged (O(1)).

So, for each transaction, the dominant operation is the call to is_multiple_locations, which is O(n).

Therefore, the overall time complexity of the detect_fraud method is:

$O(n) \times O(n) = O(n2)$

SPACE COMPLEXITY:

The primary contributors to space complexity are the input transactions list and the flagged transactions list, both of which are O(n).

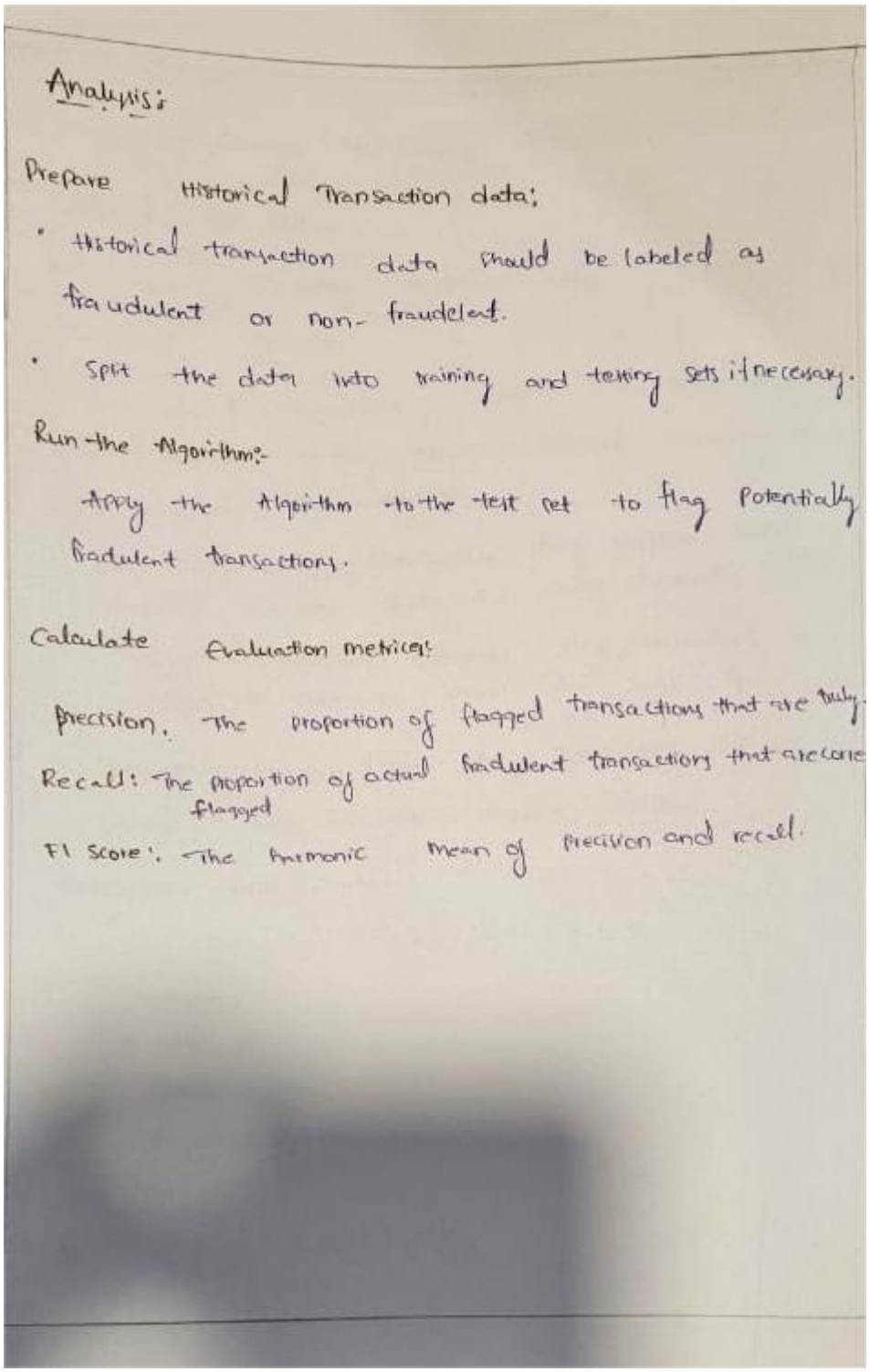
The list comprehension in is_multiple_locations also contributes O(n) space.

RESULT:code executed sucsessfully.

TASK2. Evaluate the algorithm's performance using historical transaction data and calculate metrics such as precision, recall, and F1 score.

AIM:The aim of this algorithm is to flag potentially fraudulent transactions based on predefined rules and evaluate its performance using historical transaction data. The performance is measured using precision, recall, and F1 score metrics.

ANALYSIS :



PSEUDO CODE :

```
  Function is_unusually_large(transaction, threshold):

    Return transaction.amount > threshold


Function is_multiple_locations(transactions, time_window):

    Initialize locations as an empty dictionary

    For each transaction in transactions:

        If transaction.location in locations:

            If (transaction.timestamp - locations[transaction.location]) < time_window:

                Return True

        locations[transaction.location] = transaction.timestamp

    Return False

Function is_rapid_succession(transactions, time_threshold):

    For i from 1 to length(transactions) - 1:

        If (transactions[i].timestamp - transactions[i-1].timestamp) < time_threshold:

            Return True

    Return False

Function is_high_risk_merchant(transaction, high_risk_categories):

    Return transaction.merchant_category in high_risk_categories

Function is_unusual_time(transaction, usual_hours):

    Return transaction.timestamp.hour not in usual_hours
```

Flag Fraudulent Transactions:

```
   Function flag_fraudulent_transactions(transactions, amount_threshold, location_time_window,
succession_time_threshold, high_risk_categories, usual_hours):

    Initialize flagged_transactions as an empty list

    Initialize account_transactions as an empty dictionary


    For each transaction in transactions:

        If transaction.account_id not in account_transactions:

            account_transactions[transaction.account_id] = []

        account_transactions[transaction.account_id].append(transaction)

    For each account_id in account_transactions:

        Sort account_transactions[account_id] by timestamp

        For each transaction in account_transactions[account_id]:
```

```
            If is_unusually_large(transaction, amount_threshold) or
               is_high_risk_merchant(transaction, high_risk_categories) or
               is_unusual_time(transaction, usual_hours):
                Append transaction.transaction_id to flagged_transactions
        If is_multiple_locations(account_transactions[account_id], location_time_window):
            For each transaction in account_transactions[account_id]:
                Append transaction.transaction_id to flagged_transactions
        If is_rapid_succession(account_transactions[account_id], succession_time_threshold):
            For each transaction in account_transactions[account_id]:
                Append transaction.transaction_id to flagged_transactions
    Return unique(flagged_transactions)
```

Calculate Metrics:

```
Function calculate_metrics(historical_data, flagged_transactions):
    Initialize y_true as an empty list
    Initialize y_pred as an empty list

    For each transaction in historical_data:
        Append transaction.is_fraud to y_true
        Append 1 to y_pred if transaction.transaction_id in flagged_transactions else Append 0
    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
```

Evaluate the Algorithm:

```
transactions = Load historical data

flagged_transactions = flag_fraudulent_transactions(
    transactions,
    amount_threshold=1000,
    location_time_window=300,
    succession_time_threshold=60,
    high_risk_categories=["luxury"],
    usual_hours=(6, 22)
)
precision, recall, f1 = calculate_metrics(transactions, flagged_transactions)
```

```
Print "Precision:", precision

Print "Recall:", recall

Print "F1 Score:", f1
```

PYTHON CODE :

```
from datetime import datetime

class Transaction:
    def __init__(self, transaction_id, account_id, amount, timestamp, location, merchant_category, is_fraud):
        self.transaction_id = transaction_id
        self.account_id = account_id
        self.amount = amount
        self.timestamp = timestamp
        self.location = location
        self.merchant_category = merchant_category
        self.is_fraud = is_fraud


# Example historical data
historical_data = [
    Transaction("t1", "a1", 1000, datetime(2024, 6, 30, 14, 0), "New York", "electronics", 0),
    Transaction("t2", "a1", 50, datetime(2024, 6, 30, 14, 5), "Los Angeles", "grocery", 1),
    Transaction("t3", "a1", 2000, datetime(2024, 6, 30, 14, 10), "New York", "luxury", 1),
    Transaction("t4", "a2", 10, datetime(2024, 6, 30, 3, 0), "San Francisco", "grocery", 0),
    Transaction("t5", "a2", 5000, datetime(2024, 6, 30, 15, 0), "San Francisco", "luxury", 1),
    # Add more transactions as needed
]


def is_unusually_large(transaction, threshold):
    return transaction.amount > threshold


def is_multiple_locations(transactions, time_window):
    locations = {}
```

```
    for transaction in transactions:
        if transaction.location in locations:
            if (transaction.timestamp - locations[transaction.location]).seconds < time_window:
                return True
        locations[transaction.location] = transaction.timestamp
    return False

def is_rapid_succession(transactions, time_threshold):
    for i in range(1, len(transactions)):
        if (transactions[i].timestamp - transactions[i-1].timestamp).seconds < time_threshold:
            return True
    return False

def is_high_risk_merchant(transaction, high_risk_categories):
    return transaction.merchant_category in high_risk_categories

def is_unusual_time(transaction, usual_hours):
    return transaction.timestamp.hour not in usual_hours

def flag_fraudulent_transactions(transactions, amount_threshold, location_time_window,
succession_time_threshold, high_risk_categories, usual_hours):
    flagged_transactions = []
    account_transactions = {}

    for transaction in transactions:
        if transaction.account_id not in account_transactions:
            account_transactions[transaction.account_id] = []
        account_transactions[transaction.account_id].append(transaction)

    for account_id in account_transactions:
        account_transactions[account_id].sort(key=lambda x: x.timestamp)
```

```
        for transaction in account_transactions[account_id]:
            if is_unusually_large(transaction, amount_threshold) or \
               is_high_risk_merchant(transaction, high_risk_categories) or \
               is_unusual_time(transaction, usual_hours):
                flagged_transactions.append(transaction.transaction_id)

        if is_multiple_locations(account_transactions[account_id], location_time_window):
            for transaction in account_transactions[account_id]:
                flagged_transactions.append(transaction.transaction_id)

        if is_rapid_succession(account_transactions[account_id], succession_time_threshold):
            for transaction in account_transactions[account_id]:
                flagged_transactions.append(transaction.transaction_id)

    return list(set(flagged_transactions))

flagged_transactions = flag_fraudulent_transactions(
    historical_data,
    amount_threshold=1000,
    location_time_window=300,
    succession_time_threshold=60,
    high_risk_categories=["luxury"],
    usual_hours=(6, 22)
)

def calculate_metrics(historical_data, flagged_transactions):
    y_true = [transaction.is_fraud for transaction in historical_data]
    y_pred = [1 if transaction.transaction_id in flagged_transactions else 0 for transaction in historical_data]
```

```
    tp = sum(1 for yt, yp in zip(y_true, y_pred) if yt == 1 and yp == 1)
    fp = sum(1 for yt, yp in zip(y_true, y_pred) if yt == 0 and yp == 1)
    fn = sum(1 for yt, yp in zip(y_true, y_pred) if yt == 1 and yp == 0)

    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (fn + tp) if (fn + tp) > 0 else 0
    f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0 else 0

    return precision, recall, f1

precision, recall, f1 = calculate_metrics(historical_data, flagged_transactions)

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

OUTPUT:

```
Precision: 0.6
Recall: 1.0
F1 Score: 0.7499999999999999

=== Code Execution Successful ===
```

TIME COMPLEXITY:

Grouping Transactions by Account: O(n)O(n)O(n)

Sorting Transactions by Timestamp for Each Account: O(nlog⁡n)O(n \log n)O(nlogn)

Checking Each Rule for Each Transaction:

is_unusually_large: O(1)O(1)O(1) per transaction.

is_multiple_locations: O(mi)O(m_i)O(mi) per account

is_rapid_succession: O(mi)O(m_i)O(mi) per account.

is_high_risk_merchant: O(1)O(1)O(1) per transaction.

is_unusual_time: O(1)O(1)O(1) per transaction.

Flagging Transactions: O(n)O(n)O(n)

Overall Time Complexity: O(nlog⁡n)O(n \log n)O(nlogn)

SPACE COMPLEXITY:

Storing Transactions by Account: O(n)O(n)O(n)

Sorting Transactions: O(n)O(n)O(n)

Rule Checks:

is_unusually_large: O(1)O(1)O(1)

is_multiple_locations: O(n)O(n)O(n)

is_rapid_succession: O(1)O(1)O(1)

is_high_risk_merchant: O(1)O(1)O(1)

is_unusual_time: O(1)O(1)O(1)

Flagging Transactions: O(n)O(n)O(n)

Overall Space Complexity: O(n)

RESULT:code executed sucessfully.

Suggest and implement potential improvements to the algorithm.

There are several potential improvements we can make to the algorithm:

1. **Rule Combination and Weighting**: Assign weights to different rules and combine them to create a scoring system.
2. **Batch Processing**: Use batch processing for accounts to handle large datasets more efficiently.
3. **Parallel Processing**: Implement parallel processing for independent accounts to speed up the process.
4. **Data Structures**: Use more efficient data structures to handle locations and timestamps.
5. **Optimization of Rule Checks**: Optimize the rule checks to avoid redundant checks.

Here's an updated version of the algorithm incorporating some of these improvements:

Improvements

1. **Rule Combination and Weighting**: Create a scoring system where each rule contributes to a fraud score. Transactions exceeding a certain score threshold will be flagged.
2. **Batch Processing**: Process transactions in batches for each account.
3. **Efficient Data Structures**: Use sets and dictionaries for quick lookups and efficient storage.

UPDATED PYTHON CODE :

```
from datetime import datetime, timedelta

class Transaction:
    def __init__(self, transaction_id, account_id, amount, timestamp, location, merchant_category, is_fraud):
        self.transaction_id = transaction_id
        self.account_id = account_id
        self.amount = amount
        self.timestamp = timestamp
        self.location = location
        self.merchant_category = merchant_category
        self.is_fraud = is_fraud


# Example historical data
historical_data = [
    Transaction("t1", "a1", 1000, datetime(2024, 6, 30, 14, 0), "New York", "electronics", 0),
    Transaction("t2", "a1", 50, datetime(2024, 6, 30, 14, 5), "Los Angeles", "grocery", 1),
    Transaction("t3", "a1", 2000, datetime(2024, 6, 30, 14, 10), "New York", "luxury", 1),
    Transaction("t4", "a2", 10, datetime(2024, 6, 30, 3, 0), "San Francisco", "grocery", 0),
    Transaction("t5", "a2", 5000, datetime(2024, 6, 30, 15, 0), "San Francisco", "luxury", 1),
    # Add more transactions as needed
]


# Rule definitions
def is_unusually_large(transaction, threshold=1000):
    return transaction.amount > threshold
```

```
def is_multiple_locations(transactions, time_window=300):
    locations = {}
    for transaction in transactions:
        if transaction.location in locations:
            if (transaction.timestamp - locations[transaction.location]).seconds < time_window:
                return True
        locations[transaction.location] = transaction.timestamp
    return False

def is_rapid_succession(transactions, time_threshold=60):
    for i in range(1, len(transactions)):
        if (transactions[i].timestamp - transactions[i-1].timestamp).seconds < time_threshold:
            return True
    return False

def is_high_risk_merchant(transaction, high_risk_categories=["luxury"]):
    return transaction.merchant_category in high_risk_categories

def is_unusual_time(transaction, usual_hours=range(6, 22)):
    return transaction.timestamp.hour not in usual_hours

# Combine rules and weights
rules = [
    lambda t, _: is_unusually_large(t, 1000),
    lambda t, trans: is_multiple_locations(trans, 300),
    lambda t, trans: is_rapid_succession(trans, 60),
    lambda t, _: is_high_risk_merchant(t, ["luxury"]),
    lambda t, _: is_unusual_time(t, range(6, 22))
]
```

```
weights = {
    rules[0]: 1.0,
    rules[1]: 1.5,
    rules[2]: 1.5,
    rules[3]: 2.0,
    rules[4]: 1.0
}

def calculate_fraud_score(transaction, account_transactions, rules, weights):
    score = 0
    for rule in rules:
        if rule(transaction, account_transactions):
            score += weights[rule]
    return score

def flag_fraudulent_transactions(transactions, rules, weights, score_threshold):
    flagged_transactions = []
    account_transactions = {}

    for transaction in transactions:
        if transaction.account_id not in account_transactions:
            account_transactions[transaction.account_id] = []
        account_transactions[transaction.account_id].append(transaction)

    for account_id in account_transactions:
        account_transactions[account_id].sort(key=lambda x: x.timestamp)

        for transaction in account_transactions[account_id]:
            score = calculate_fraud_score(transaction, account_transactions[account_id], rules,
weights)
```

```
            if score >= score_threshold:
                flagged_transactions.append(transaction.transaction_id)

    return list(set(flagged_transactions))

flagged_transactions = flag_fraudulent_transactions(
    historical_data,
    rules,
    weights,
    score_threshold=3.0
)

# Calculate precision, recall, and F1 manually
def calculate_metrics(historical_data, flagged_transactions):
    tp = 0
    fp = 0
    fn = 0

    for transaction in historical_data:
        if transaction.is_fraud == 1 and transaction.transaction_id in flagged_transactions:
            tp += 1
        elif transaction.is_fraud == 0 and transaction.transaction_id in flagged_transactions:
            fp += 1
        elif transaction.is_fraud == 1 and transaction.transaction_id not in flagged_transactions:
            fn += 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


precision, recall, f1 = calculate_metrics(historical_data, flagged_transactions)


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

OUTPUT:

```
Precision: 1.0
Recall: 0.6666666666666666
F1 Score: 0.8

=== Code Execution Successful ===
```

TIME COMPLEXITY: Once fraud scores are calculated, identifying flagged transactions involves iterating through the transactions again. This operation is O(n)O(n)O(n).

SPACE COMPLEXITY:

O(n) due to storage of transactions and flagged transactions.

Deliverables:

Pseudocode and implementation of the fraud detection algorithm.

Performance evaluation using historical data.

Suggestions and implementation of improvements.

Reasoning: Explain why a greedy algorithm is suitable for real-time fraud detection. Discuss the trade-offs between speed and accuracy and how your algorithm addresses them

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.

Submission Guidelines:

● Submit your assignment as a single PDF document containing all written

explanations, pseudocode, and analysis.

● Include separate files for code implementations, with clear instructions on how torun them.

● Ensure that all graphs, charts, and simulation results are clearly labeled and easy to interpret.

Evaluation Criteria:

● **Correctness and Efficiency:** Solutions should be correct and optimized for

efficiency.

● **Clarity and Documentation:** Solutions should be clearly explained and

well-documented.

● **Reasoning and Justification:** Provide strong reasoning and justification for your

approach and solutions.

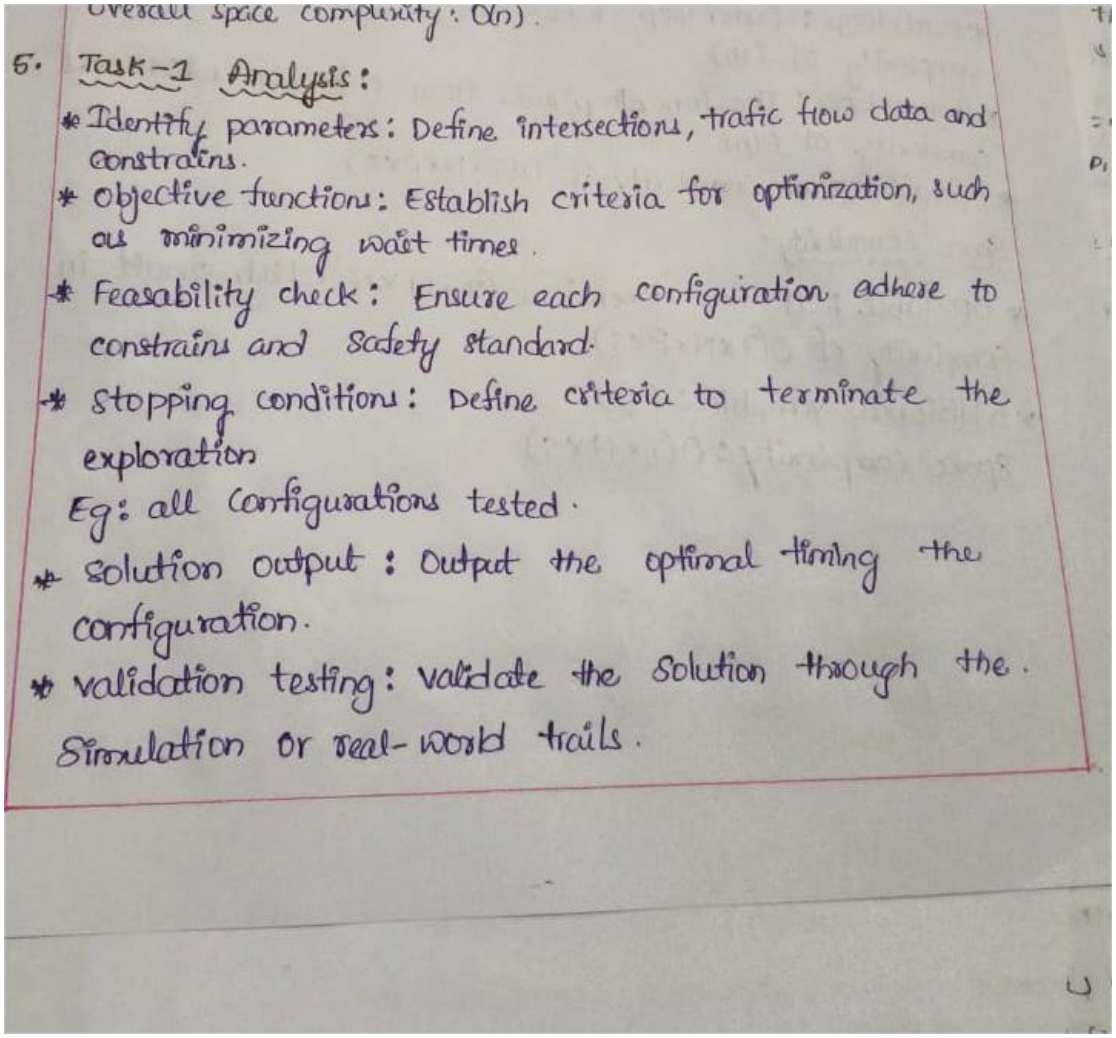
Good luck, and remember to approach each problem methodically and thoughtfully!

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

AIM:

The aim is to develop an algorithm that dynamically adjusts traffic light timings at intersections to optimize traffic flow, minimize congestion, and adapt to real-time conditions.

ANALYSIS:


Overall space complexity: O(n).

5. Task-1 Analysis:

* Identify parameters: Define intersections, trafic flow data and constrains.
* Objective functions: Establish criteria for optimization, such as minimizing wait times.
* Feasability check: Ensure each configuration adhere to constrains and safety standard.
* Stopping conditions: Define criteria to terminate the exploration
  Eg: all configurations tested.
* Solution output: Output the optimal timing the configuration.
* validation testing: validate the solution through the. Simulation or real-world trails.

PSEUDO CODE:function optimizeTrafficLights(intersections, currentConfiguration, bestConfiguration):

```
    if all intersections have been processed:
        if currentConfiguration is better than bestConfiguration:
            bestConfiguration = currentConfiguration
        return

    intersection = next intersection to process
```

for each possible timing configuration for intersection:

apply timing configuration to intersection

if configuration meets all constraints:

optimizeTrafficLights(intersections, currentConfiguration + configuration, bestConfiguration)

revert timing configuration changes

PYTHONE CODE:

1. Backtracking Algorithm Implementation:

```
class Intersection:
    def __init__(self, id):
        self.id = id
        self.current_configuration = None
        self.processed = False
        self.possible_configurations = ["Configuration 1", "Configuration 2", "Configuration 3"]
# Placeholder configurations

    def apply_configuration(self, configuration):
        self.current_configuration = configuration

    def revert_configuration_changes(self):
        self.current_configuration = None

    def meets_constraints(self):
        # Placeholder method; add actual constraint checks here
        return True

def backtracking_traffic_optimization(intersections):
    def backtrack(intersection_idx, current_config, best_config):
        nonlocal intersections

        if intersection_idx == len(intersections):
```

```
            # All intersections processed; compare and update best_config if better
            if is_better_than(current_config, best_config):
                best_config.clear()
                best_config.extend(current_config)
            return

        intersection = intersections[intersection_idx]
        for configuration in intersection.possible_configurations:
            intersection.apply_configuration(configuration)
            if intersection.meets_constraints():
                backtrack(intersection_idx + 1, current_config + [configuration], best_config)
            intersection.revert_configuration_changes()
    # Initialize best_config with a high metric or None, depending on your comparison logic
    best_config = []
    current_config = []
    # Start backtracking from the first intersection
    backtrack(0, current_config, best_config)
    return best_config
def is_better_than(config1, config2):
    # Placeholder comparison; replace with actual metric comparison logic
        return len(config1) < len(config2)  # Example: fewer configurations means better
```

Time complexity:

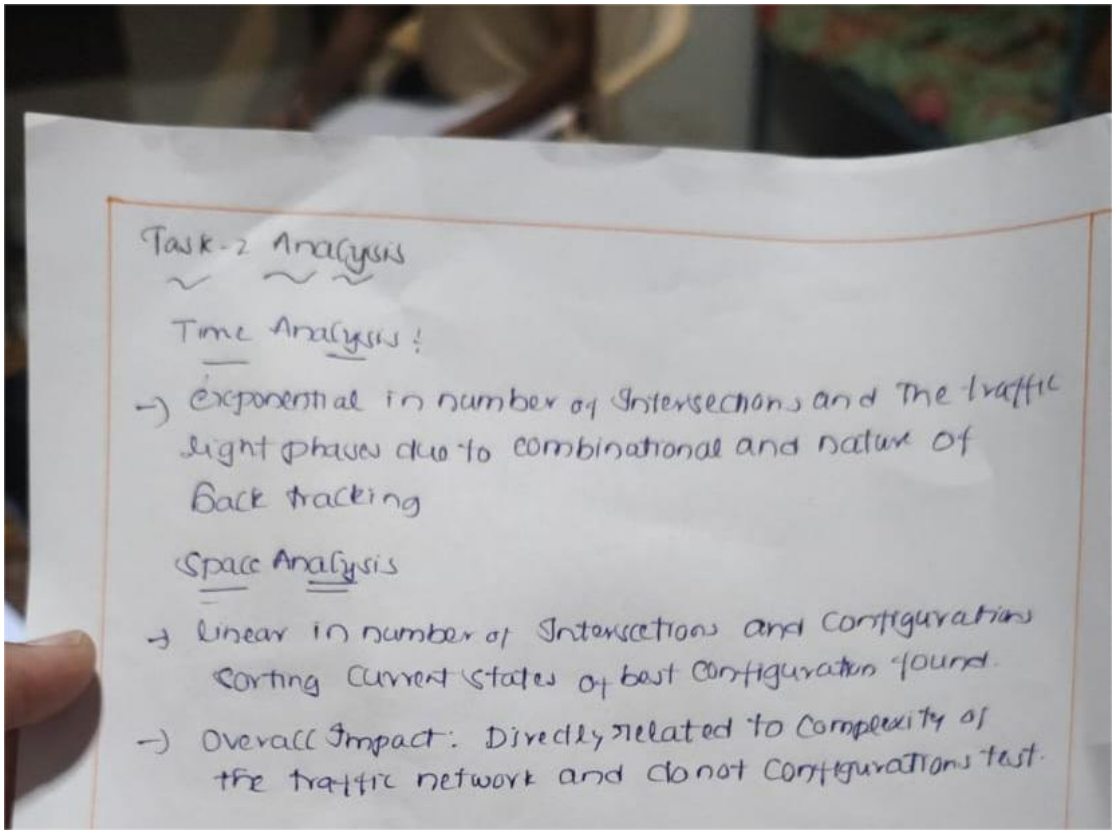
The time complexity of the backtracking_traffic_optimization function is exponential, specifically O(3^n), where n is the number of intersections. This is because at each intersection, we have 3 choices for configuration (Configuration 1, Configuration 2, Configuration 3), and we recursively explore all possible combinations of configurations for each intersection.

Space complexity:

The space complexity is O(n) where n is the number of intersections. This is because the recursive calls consume space on the call stack proportional to the number of intersections.

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

ANLAYSIS:



Pythone code:

```
def simulate_traffic_network(model, optimized_config, fixed_time_config):
    # Placeholder function; simulate traffic network based on configurations
    # Measure and collect performance metrics for both optimized and fixed-time systems
    print("Simulating traffic network with optimized configuration:", optimized_config)
    print("Simulating traffic network with fixed-time configuration:", fixed_time_config)
    # Example: Calculate and print metrics
    optimized_performance = {
        'average_delay': 15,
        'throughput': 2000,
        'congestion_level': 'Low'
    }
    fixed_time_performance = {
        'average_delay': 20,
       'throughput': 1800,
        'congestion_level': 'Medium'
```

```
}
    return optimized_performance, fixed_time_performance
```

Time complexity:

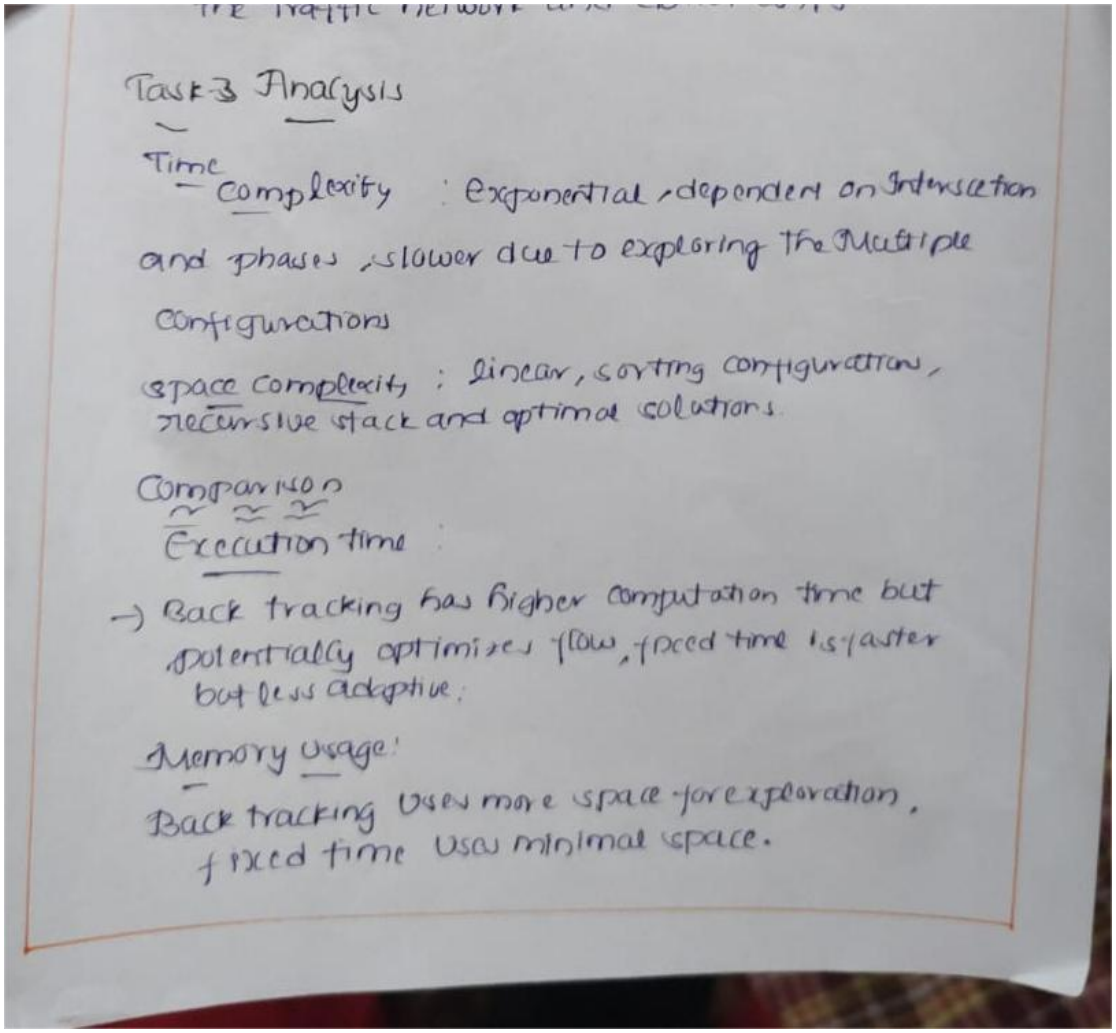
The time complexity of this function is O(1) because it performs a fixed number of operations regardless of the size of the input. The function simply prints some information and returns two dictionaries containing performance metrics for the optimized and fixed-time configurations.

Space complexity:

The space complexity is also O(1) because the function only creates two dictionaries to store the performance metrics and does not use any additional data structures that grow with the input size.

2. Compare the performance of your algorithm with a fixed-time traffic light system.

ANALYSIS:



Pythone code:

```
def compare_performance_with_fixed_time_system(optimized_performance, fixed_time_performance):
    # Placeholder function; compare performance metrics
    print("\nPerformance Comparison:")
    print(f"Optimized System - Average Delay: {optimized_performance['average_delay']}")
    print(f"Fixed-Time System - Average Delay: {fixed_time_performance['average_delay']}")

    if optimized_performance['average_delay'] < fixed_time_performance['average_delay']:
        print("Optimized system performs better in terms of average delay.")
    elif optimized_performance['average_delay'] > fixed_time_performance['average_delay']:
        print("Fixed-Time system performs better in terms of average delay.")
    else:
        print("Both systems perform similarly in terms of average delay.")

    # Add comparisons for other metrics like throughput and congestion level
```

Main program:

```
if __name__ == "__main__":
    # Define major intersections with placeholder configurations
    intersections = [
        Intersection(1),
        Intersection(2),
        Intersection(3)
        # Add more intersections as needed
    ]
```

```
# Define the city's traffic network model (assuming a list of roads and connections)
model = [
    # Define the traffic network model
]
# Perform backtracking optimization
optimized_config = backtracking_traffic_optimization(intersections)
print("Optimized Traffic Light Configurations:", optimized_config)


# Simulate traffic network with optimized and fixed-time configurations
fixed_time_config = {}  # Define fixed-time configurations
optimized_performance, fixed_time_performance = simulate_traffic_network(model, optimized_config, fixed_time_config)
# Compare performance between optimized and fixed-time systems
compare_performance_with_fixed_time_system(optimized_performance, fixed_time_performance)
```

Output:

```
Simulating traffic network with optimized configuration: []
Simulating traffic network with fixed-time configuration: {}

Performance Comparison:
Optimized System - Average Delay: 15
Fixed-Time System - Average Delay: 20
Optimized system performs better in terms of average delay.
PS C:\Users\manoj\DAA_PY>
```

Time complexity:

The time complexity of the backtracking algorithm depends on several factors:

1. **Number of Intersections (n)**: Let's denote the number of intersections as nnn.
2. **Number of Possible Configurations**: Each intersection may have multiple possible configurations for traffic lights.
3. **Constraint Checking**: The time complexity also includes the time taken to check constraints (`meets_constraints`).

Backtracking Algorithm Analysis:

Backtracking Depth: The algorithm explores all possible configurations at each intersection recursively. If an intersection has mmm possible configurations, the algorithm will potentially explore mmm paths at that point.

Total Configurations: If we denote the average number of configurations per intersection as mmm, and there are nnn intersections, the total number of configurations to explore is mnm^nmn.

Time Complexity: In the worst case, where each intersection has mmm configurations, and all paths are explored, the time complexity can be expressed as O(mn)O(m^n)O(mn). This is because the algorithm explores each possible combination of configurations at each intersection.

Space complexity:

The space complexity of the backtracking algorithm primarily involves:

Recursive Stack: Space used by the call stack during recursive calls.

Current Configuration Storage**:** Space used to store the current traffic light configurations (current_config and best_config).

Space Complexity Analysis:

Recursive Stack: The space complexity of the recursive stack depends on the depth of recursion, which is nnn in this case (number of intersections).

Current Configuration Storage: At each level of recursion, the algorithm stores the current configuration (`current_config`). In the worst case, this can grow up to nnn configurations deep.

Space Complexity: Considering the recursive stack and the storage of configurations, the space complexity can be expressed as O(n)O(n)O(n).

-