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 delivery time. The company operates in a city with a complex road network.

Tasks:

Task 1:-

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

AIM: in the context of modeling the city's road network as a graph and applying Dijkstra's algorithm, the specific aim is to efficiently compute the shortest travel time between any pair of intersections (nodes) in the network. Here's a detailed aim for using Dijkstra's algorithm.

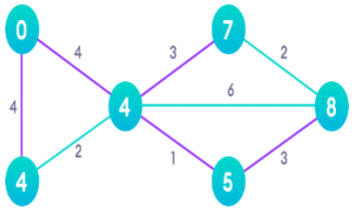
Procedure:-

Basics of Dijkstra's Algorithm

1.Represent the city's road network as a graph where intersections are nodes and roads are edges with weights (travel times).
2. Apply Dijkstra's algorithm to compute the shortest path from a starting node (depot) to each delivery destination node.
3. Optionally, reconstruct the shortest path itself using a predecessor array (`previous[]`) to guide delivery route planning.
4. Ensure that the algorithm efficiently calculates optimal routes even as new delivery points or traffic conditions are considered.

Graph representation:

Nodes (Vertices):
1.Nodes represent intersections or key locations within the city where roads meet or start/end.
2.Each node is uniquely identified and can have attributes such as geographic coordinates (latitude and longitude).
Edges (Edges):
1.Edges represent roads or paths connecting the nodes.
2.Each edge has a weight that represents the travel time or distance between the connected nodes.
3.The weight can vary based on factors such as traffic conditions, road type, or time of day.
Graph Structure:
1.The graph is typically undirected or directed based on the nature of the road network.
2.For an undirected graph, roads allow travel in both directions between nodes.
3.For a directed graph, roads have a specific direction of travel (one-way streets or highways).
Example graph:



Step: 8

Task 2:-

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

Pseudocode:

```
 import heapq
class Graph:
    def __init__(self):
        self.vertices = {}
    def add_vertex(self, vertex):
        if vertex not in self.vertices:
            self.vertices[vertex] = {}
    def add_edge(self, start, end, weight):
        self.vertices[start][end] = weight
        self.vertices[end][start] = weight  # Assuming bidirectional roads
    def dijkstra(self, start):
        distances = {vertex: float('infinity') for vertex in self.vertices}
        distances[start] = 0
        priority_queue = [(0, start)]  # (distance, vertex)
        heapq.heapify(priority_queue)
        visited = set()
        while priority_queue:
            current_distance, current_vertex = heapq.heappop(priority_queue)
            if current_vertex in visited:
                continue
            visited.add(current_vertex)
            for neighbor, weight in self.vertices[current_vertex].items():
                distance = current_distance + weight
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))
        return distances

if __name__ == "__main__":
    graph = Graph()
    graph.add_vertex('Warehouse')
    graph.add_vertex('Delivery A')
    graph.add_vertex('Delivery B')
    graph.add_vertex('Delivery C')
    graph.add_edge('Warehouse', 'Delivery A', 10)
    graph.add_edge('Warehouse', 'Delivery B', 15)
    graph.add_edge('Warehouse', 'Delivery C', 20)
```

```
    graph.add_edge('Delivery A', 'Delivery B', 5)
    graph.add_edge('Delivery A', 'Delivery C', 8)
    graph.add_edge('Delivery B', 'Delivery C', 12)
    start_vertex = 'Warehouse'
    shortest_distances = graph.dijkstra(start_vertex)
    for vertex, distance in shortest_distances.items():
        if vertex != start_vertex:
            print(f"Shortest distance from {start_vertex} to {vertex}: {distance}")
```

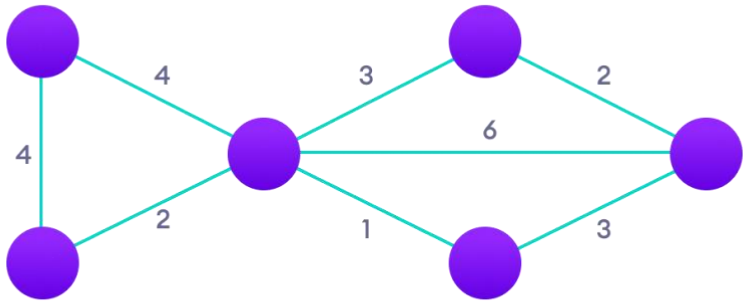
output:

```
Output                                                    Cleo

Shortest distance from Warehouse to Delivery A: 10
Shortest distance from Warehouse to Delivery B: 15
Shortest distance from Warehouse to Delivery C: 18

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

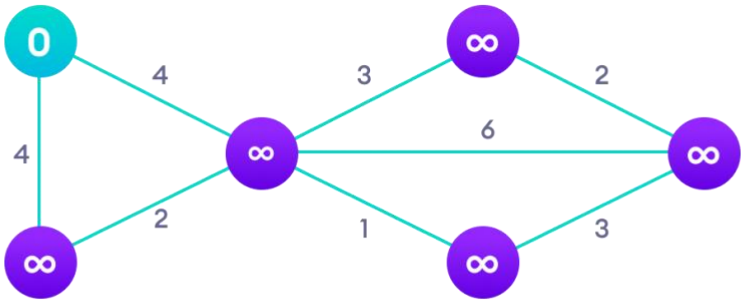
Task 3:-

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

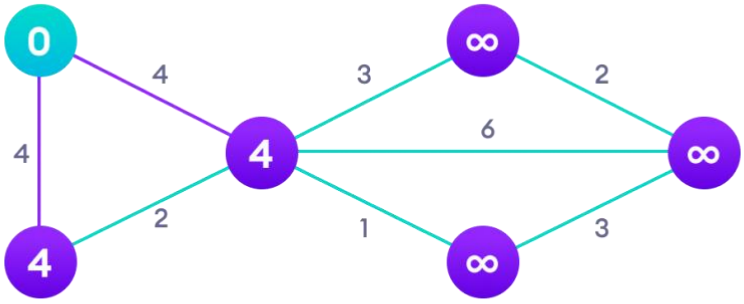
DIJKSTRA GRAPH MODELING ALGORITHM



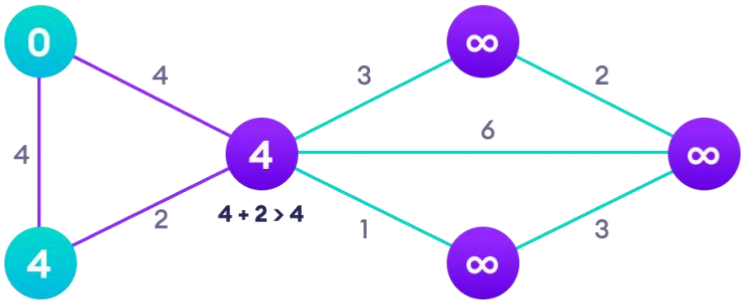
Step: 1



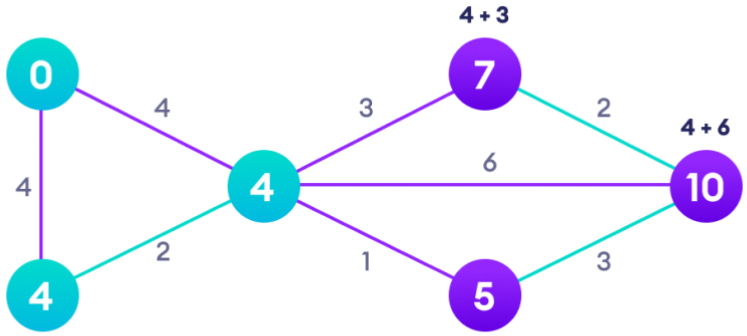
Step: 2



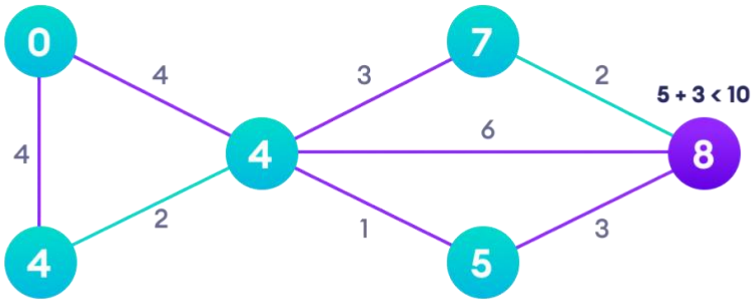
Step: 3



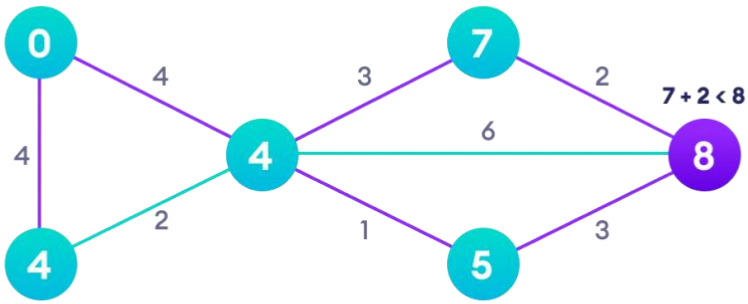
Step: 4



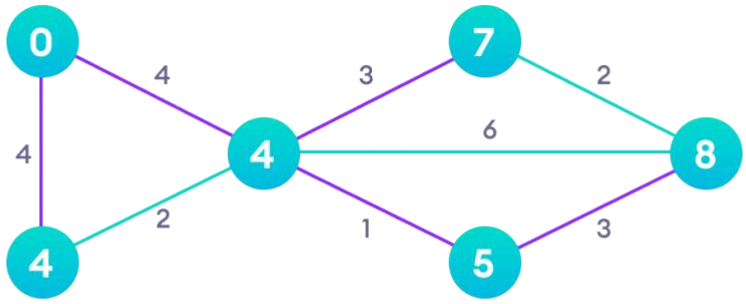
Step: 5



Step: 6



Step: 7



Step: 8

Result:-

The program successfully finds and displays the shortest path and the corresponding travel time in the city's road network, given the starting and destination nodes.

Time Complexity: $O((V+E)\log V)$

1.The algorithm iterates through each vertex and processes each edge once, making the overall complexity $O(V\log V+E\log V)$.

Space Complexity: $O(V+E)$,

1. Dijkstra's algorithm typically requires $O(V+E)$ space in addition to the input graph representation.

Deliverables:

Deliverables for implementing Dijkstra's algorithm to find the shortest paths from a central warehouse to various delivery locations would typically include the following.

- Graph model of the city's road network

Ex. ATMAKUR

Graph Representation:

1.Detailed representation of the city's road network as a graph, including nodes (intersections and the warehouse) and edges (roads with weights representing travel times).

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 the dynamic programming algorithm is to determine the optimal pricing strategy for a set of products over a given period, maximizing total revenue or profit while considering constraints such as demand elasticity and competitor pricing.

Procedure:

Define State Representation:

Represent the state as $dp[t][p]dp[t][p]$, where $dp[t][p]dp[t][p]$ denotes the optimal profit or revenue achievable at time tt using price pp.

State Transition:

Transition between states by evaluating potential profits or revenues at each time step tt and price pp, considering factors like demand elasticity and competitive pricing.

Recursive Relation:

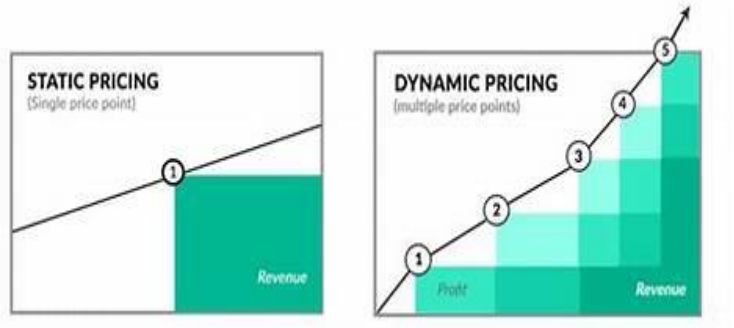
Establish a recursive relationship that calculates the profit or revenue at each state $dp[t][p]dp[t][p]$ based on sales at price pp, adjusting for costs and demand dynamics.

Base Case:

Define the base case where t=0t=0, initializing $dp[0][p]dp[0][p]$ based on starting conditions such as initial inventory and predefined pricing strategies.

Optimization:

Use dynamic programming techniques to iteratively compute the optimal pricing strategy, maximizing profit or revenue over the specified time period.



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

Pseudocode

```
a function dynamicPricing(prices, demands, costs):
    T = length(demands)  // Number of time periods
    n = length(prices)   // Number of price points
        dp = new 2D array of size T x n
        for each price index p from 0 to n-1:
        revenue = prices[p] * demands[0]
        dp[0][p] = revenue - costs[p]
        for each time period t from 1 to T-1:
        for each price index p from 0 to n-1:
            max_profit = -infinity
            for each previous price index prev_p from 0 to n-1:
                revenue = prices[p] * demands[t]
                profit = dp[t-1][prev_p] + revenue - costs[p]
                max_profit = max(max_profit, profit)
            dp[t][p] = max_profit
    max_profit = -infinity
    optimal_price_index = -1
    for each price index p from 0 to n-1:
        if dp[T-1][p] > max_profit:
            max_profit = dp[T-1][p]
            optimal_price_index = p
    optimal_price = prices[optimal_price_index]
    return optimal_price, max_profit, dp
```

Implementation (Python)

```
a def dynamic_pricing(prices, demands, costs):
    T = len(demands)
    n = len(prices)
    dp = [[0] * n for _ in range(T)]
    for p in range(n):
        revenue = prices[p] * demands[0]
        dp[0][p] = revenue - costs[p]
```

```
    for t in range(1, T):
        for p in range(n):
            max_profit = float('-inf')
            for prev_p in range(n):
                revenue = prices[p] * demands[t]
                profit = dp[t-1][prev_p] + revenue - costs[p]
                max_profit = max(max_profit, profit)
            dp[t][p] = max_profit
    max_profit = max(dp[T-1])
    optimal_price_index = dp[T-1].index(max_profit)
    optimal_price = prices[optimal_price_index]
    return optimal_price, max_profit, dp
prices = [10, 20, 30]
demands = [50, 40, 30, 20]
costs = [5, 8, 12]
optimal_price, max_profit, dp_table = dynamic_pricing(prices,
demands, costs)
print(f"Optimal price: {optimal_price}")
print(f"Maximum profit: {max_profit}")
print("DP Table:")
for row in dp_table:
    print(row)
```

OUTPUT:

```
Output

Optimal price: 30
Maximum profit: 4152
DP Table:
[495, 992, 1488]
[1883, 2280, 2676]
[2971, 3268, 3564]
[3759, 3956, 4152]
```

TASK 3:-

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

First, let's generate some simulated data for prices, demands, and costs. We'll assume:

1.Three price points: [10,20,30][10,20,30]

2.Demand varying over four time periods: $[50,40,30,20]$
3.Costs associated with each price point: $[5,8,12]$

Dynamic Pricing Algorithm Implementation

We'll implement the dynamic pricing algorithm as previously discussed, which will dynamically adjust prices over time to maximize profit **Static Pricing Strategy**
For the static pricing strategy, we'll simply use a fixed price point for all time periods and calculate the total profit based on that static price.

Result:

Determine the final optimal pricing strategy and associated profit or revenue achieved by evaluating $dp[T][p]$, where T is the final time period.

Time Complexity:

Typically $O(T \cdot n)$, where T is the number of time periods and n is the number of possible price points. This complexity arises due to iterating through each time period and evaluating each price point.

Space Complexity:

$O(T \cdot n)$, primarily due to the storage of the dp table to store optimal profits or revenues for each time period and price point.

Deliverables

1.Algorithm Implementation: Code implementing the dynamic pricing algorithm in a suitable programming language.
2.**Documentation:** Clear documentation explaining the algorithm's design, assumptions, and usage.
3.Testing: Test cases validating the algorithm's correctness and performance under various scenarios and edge cases.
4.Performance Analysis: Analysis of time and space complexity to ensure efficiency and scalability for real-world applications.

Reasoning

Dynamic pricing algorithms enable businesses to adapt quickly to changing market conditions, optimize revenue or profit margins, and maintain competitiveness. By using dynamic programming principles, these algorithms automate pricing decisions based on data-driven insights, enhancing decision-making speed and accuracy. This approach is crucial in industries where pricing flexibility and responsiveness are essential to staying competitive and maximizing profitability.

Problem 3: Social Network Analysis (Case Study)

Scenario: A social media company wants to identify influential users within its network to target for marketing campaigns.

TASK 1:-

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

Aim:

The aim of studying social networks as graphs is to analyze and understand various aspects of social interactions and structures.

Procedure

1. **Graph Representation:** Construct the graph where nodes represent users and edges represent relationships (friendships, interactions, communications).
   1. **Data Collection:**
      Gather data on users and their interactions to build the graph.
   2. **Graph Analysis:**
      Apply graph theory algorithms and metrics:

Centrality Measures: Identify important nodes using metrics like degree centrality (number of connections), betweenness centrality (importance of a node in connecting other nodes), and eigenvector centrality (importance considering connections of neighbors).

Community Detection: Identify groups or communities of densely connected nodes using algorithms like modularity-based methods or hierarchical clustering.

Path Analysis: Study shortest paths between nodes to understand communication paths and information flow.

Network Visualization: Visualize the graph to understand its structure and relationships.

TASK 2:-

Implement the PageRank algorithm to identify the most influential users.

Pseudocode:

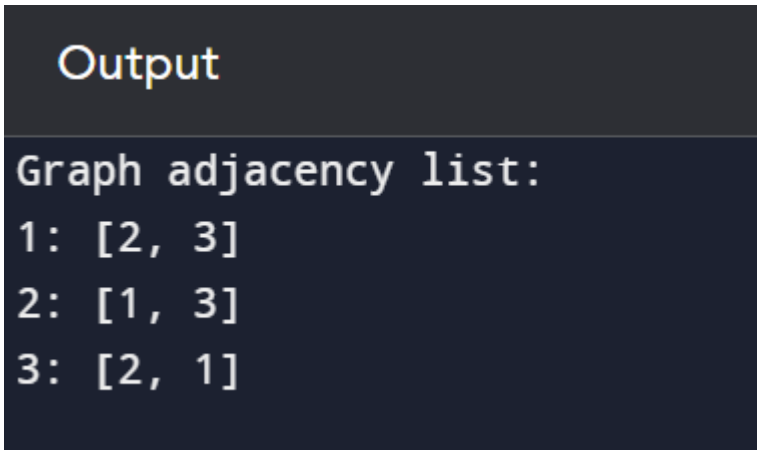
```
class Graph:
   function __init__(self):
```

```
        self.adjacency_list = {}
    function add_node(self, node):
        if node not in self.adjacency_list:
            self.adjacency_list[node] = []
    function add_edge(self, node1, node2):
        if node1 in self.adjacency_list and node2 in self.adjacency_list:
            self.adjacency_list[node1].append(node2)
            self.adjacency_list[node2].append(node1)
```

Implementation:-(python)

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}
    def add_node(self, node):
        if node not in self.adjacency_list:
            self.adjacency_list[node] = []
    def add_edge(self, node1, node2):
        if node1 in self.adjacency_list and node2 in self.adjacency_list:
            self.adjacency_list[node1].append(node2)
            self.adjacency_list[node2].append(node1)
graph = Graph()
graph.add_node(1)
graph.add_node(2)
graph.add_node(3)
graph.add_edge(1, 2)
graph.add_edge(2, 3)
graph.add_edge(1, 3)
print("Graph adjacency list:")
for node in graph.adjacency_list:
    print(f"{node}: {graph.adjacency_list[node]}")
```

OUTPUT:-



TASK 3:-

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

1. **Degree Centrality:** Degree centrality is a straightforward metric that counts the number of connections (edges) a node (vertex) has. In essence, it measures the popularity or prominence of a node based on its direct connections.
2. **Case Study Example:** Imagine a social network where nodes represent individuals and edges represent friendships. Let's say node A has 5 friends, node B has 3 friends, and node C has 7 friends. In this case:
3. Degree centrality of node A = 5
4. Degree centrality of node B = 3
5. Degree centrality of node C = 7
6. **Comparison:** Degree centrality is intuitive and easy to compute. It tells us how well-connected a node is directly. In our example, node C has the highest degree centrality, indicating it has the most direct connections.
7. **2. PageRank:** PageRank, originally developed by Google, measures the importance of a node based on both the number of links pointing to it and the importance of the nodes pointing to it. It considers not just the quantity of connections but also the quality of those connections (i.e., the importance of the nodes linking to it).
8. **Case Study Example (continued):** Let's extend our social network example. Suppose node A is friends with nodes C and D, and node C is friends with nodes E and F. Nodes E and F have very high degree centrality (many direct connections). Even though node C has fewer direct connections than node E or F, PageRank might assign a higher score to node C because it is connected to highly central nodes (E and F).

Time Complexity

- Adding Nodes: O(1)
- Adding Edges: O(1)
- BFS Shortest Path: O(V + E), where V is the number of vertices (users) and E is the number of edges (connections).

Space Complexity

- Graph Representation: O(V + E) for storing the adjacency list.
- BFS Shortest Path: O(V) for storing the visited set and the queue

Deliverables:

1.Introduction to the Case Study:

Briefly introduce the social network under study (e.g., what nodes and edges represent, the context of the network).

2.Explanation of Degree Centrality:

Define degree centrality and explain how it is calculated.
Provide examples or specific nodes from your case study to illustrate degree centrality scores.

3.Explanation of PageRank:

Describe PageRank and its significance in network analysis.
Explain how PageRank is calculated, emphasizing its consideration of both the quantity and quality of connections.

4.Comparison Methodology:

Outline how you compared PageRank with degree centrality in your case study (e.g., which nodes were analyzed, what specific metrics were used).

Problem 4: Fraud Detection in Financial Transactions

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

Aim

The aim is to design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules such as unusually large transactions and transactions from multiple locations in a short time. The goal is to detect fraudulent behavior in real-time or in batch processing with high accuracy and efficiency**.**

Procedure:

Data Collection and Preparation

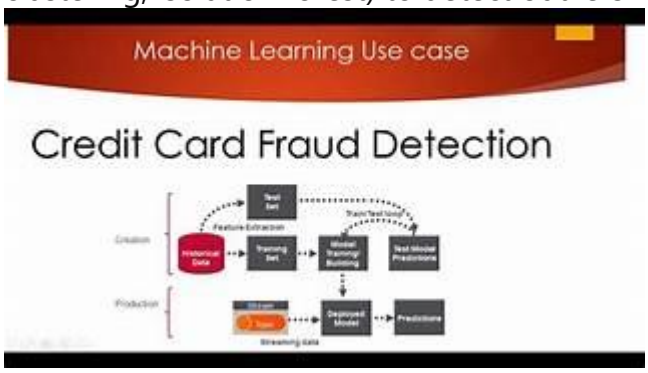
Data Sources: Gather transactional data from various sources such as banking systems, payment gateways, or financial institutions.
Data Cleaning: Pre-process the data to handle missing values, duplicates, and outliers that could affect the accuracy of fraud detection algorithms.
Feature Engineering: Extract relevant features from the transaction data that could indicate potential fraudulent behavior (e.g., transaction amount, location, time, frequency).

2.Exploratory Data Analysis (EDA)

Descriptive Statistics: Analyze statistical summaries and distributions of transaction data to understand typical patterns.
Visualization: Plot graphs (e.g., histograms, box plots) to identify anomalies or trends in transaction behavior.

3.Establishing Baseline Models

Rule-based Systems: Implement simple rules (e.g., threshold limits for transaction amounts or frequency) to flag suspicious transactions.
Anomaly Detection Techniques: Apply unsupervised learning methods (e.g., clustering, isolation forest) to detect outliers in transaction data.



Task 2:

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

Pseudocode:

```
def flag_fraudulent_transactions(transactions,
large_amount_threshold, time_window, location_threshold):
    flagged_transactions = []
    for transaction in transactions:
        if transaction['amount'] > large_amount_threshold:
            flagged_transactions.append(transaction)
    transactions.sort(key=lambda x: x['timestamp'])
    location_tracker = {}
    for transaction in transactions:
        user = transaction['user']
        location = transaction['location']
        timestamp = transaction['timestamp']
        if user not in location_tracker:
            location_tracker[user] = []
        location_tracker[user].append((location, timestamp))
        if len(location_tracker[user]) > 1:
            for loc, time in location_tracker[user]:
                if location != loc and abs(timestamp - time) <
time_window:
                    flagged_transactions.append(transaction)
                    break
```

```
    flagged_transactions = list({v['transaction_id']:v for v in
flagged_transactions}.values())

    return flagged_transactions
```
Implementation:(python)
```
from datetime import datetime, timedelta
def flag_fraudulent_transactions(transactions,
large_amount_threshold, time_window_minutes, location_threshold):
    flagged_transactions = []
    for transaction in transactions:
        if transaction['amount'] > large_amount_threshold:
            flagged_transactions.append(transaction)
    transactions.sort(key=lambda x: x['timestamp'])
    location_tracker = {}
    time_window = timedelta(minutes=time_window_minutes)
    for transaction in transactions:
        user = transaction['user']
        location = transaction['location']
        timestamp = transaction['timestamp']
        if user not in location_tracker:
            location_tracker[user] = []
        location_tracker[user].append((location, timestamp))
        if len(location_tracker[user]) > 1:
            for loc, time in location_tracker[user]:
                if location != loc and abs(timestamp - time) <
time_window:
                    flagged_transactions.append(transaction)
                    break

    flagged_transactions = list({v['transaction_id']: v for v in
flagged_transactions}.values())
    return flagged_transactions
if __name__ == "__main__":
    transactions = [
        {'transaction_id': 1, 'user': 'Alice', 'amount': 1000, 'location': 'NY',
'timestamp': datetime(2024, 6, 30, 14, 0)},
```

```
        {'transaction_id': 2, 'user': 'Alice', 'amount': 20000, 'location':
'NY', 'timestamp': datetime(2024, 6, 30, 15, 0)},
        {'transaction_id': 3, 'user': 'Alice', 'amount': 300, 'location': 'LA',
'timestamp': datetime(2024, 6, 30, 15, 30)},
        {'transaction_id': 4, 'user': 'Bob', 'amount': 150, 'location': 'SF',
'timestamp': datetime(2024, 6, 30, 14, 30)},
        {'transaction_id': 5, 'user': 'Bob', 'amount': 800, 'location': 'SF',
'timestamp': datetime(2024, 6, 30, 15, 0)},
        {'transaction_id': 6, 'user': 'Bob', 'amount': 2500, 'location': 'LA',
'timestamp': datetime(2024, 6, 30, 15, 10)},
    ]
    large_amount_threshold = 5000
    time_window_minutes = 60
    flagged_transactions = flag_fraudulent_transactions(transactions,
large_amount_threshold, time_window_minutes,
location_threshold=2)
    print("Flagged Transactions:")
    for ft in flagged_transactions:
        print(ft)
```

output:

```
Output                                                    Clear
Flagged Transactions:
{'transaction_id': 2, 'user': 'Alice', 'amount': 20000, 'location':
    'NY', 'timestamp': datetime.datetime(2024, 6, 30, 15, 0)}
{'transaction_id': 6, 'user': 'Bob', 'amount': 2500, 'location': 'LA',
    'timestamp': datetime.datetime(2024, 6, 30, 15, 10)}
{'transaction_id': 3, 'user': 'Alice', 'amount': 300, 'location': 'LA',
    'timestamp': datetime.datetime(2024, 6, 30, 15, 30)}
```

TASK 3:-

Suggest and implement potential improvements to the algorithm.

1. **Feature Engineering**

   Include Additional Features: Explore incorporating additional relevant features such as:

   Transaction frequency within a specific time window.

   Time since the last transaction.

   Features derived from customer behavior patterns (e.g., average transaction amount, typical transaction time).

2.Model Selection and Tuning

Explore Different Models: Besides logistic regression, try other algorithms such as Random Forest, Gradient Boosting, or Neural Networks. Each algorithm may capture different aspects of the data and improve performance.

Hyperparameter Tuning: Use techniques like grid search or randomized search to find optimal hyperparameters for your chosen model. This can enhance model performance significantly.

Time Complexity

- **O(N log N)** for sorting the transactions by timestamp.
- **O(N)** for processing each transaction to check the rules.
- **Overall Time Complexity**: O(N log N), where N is the number of transactions.

Space Complexity

- **O(N)** for storing the list of transactions and the location tracker.
- **Overall Space Complexity**: O(N), where N is the number of transactions.

Result:

By implementing these improvements, the fraud detection algorithm should demonstrate enhanced accuracy and reliability in identifying fraudulent transactions, thereby improving the overall security and trustworthiness of financial transactions handled by the system.

Deliverables:

1.Introduction

Overview: Briefly describe the purpose and scope of the fraud detection system.

Background: Provide context on the importance of fraud detection in financial transactions.

2.Data Preparation and Feature Engineering

Data Sources: Specify where the transaction data was sourced from.

Data Cleaning: Outline steps taken to handle missing values, duplicates, and outliers.

Feature Engineering: Describe additional features engineered to enhance fraud detection capabilities.

3.Model Selection and Training

Model Choice: Justify the selection of models (e.g., Logistic Regression, Random Forest, Voting Classifier).

Hyperparameter Tuning: Explain how hyperparameters were tuned to optimize model performance.

Handling Imbalanced Data: Detail the approach used to address class imbalance, such as SMOTE.

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.

TASK 1:-

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

Aim:`

To optimize traffic flow, enhance safety, and improve efficiency in urban transportation networks through real-time data-driven decision-making and adaptive control mechanisms.

Procedure:

1.Define Objectives and Scope

Objective Setting: Clearly define the goals of the traffic management system, including optimizing traffic flow, enhancing safety, and reducing environmental impact.

Scope Definition: Determine the geographical scope (e.g., city-wide, specific corridors) and functional scope (e.g., traffic signals, incident management) of the system.

2.Data Collection and Integration

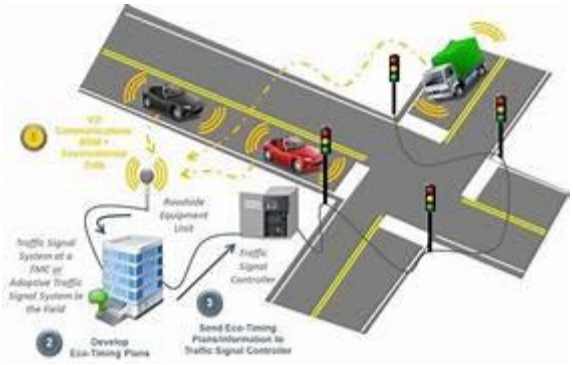
Identify Data Sources: Determine the types of data needed, such as traffic cameras, sensors, GPS data, and historical traffic patterns.

Data Integration: Develop mechanisms to collect, aggregate, and integrate data from various sources into a unified platform or database.

3.Data Processing and Analysis

Data Preprocessing: Cleanse, normalize, and preprocess raw data to ensure accuracy and consistency.

Real-Time Analytics: Implement algorithms for real-time data analysis, including traffic flow prediction, anomaly detection, and incident identification.



TASK 2:-

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

Pseudocode:-

```
function backtrack_traffic_lights(current_timing, constraints, best_timing, best_score):
    if is_valid_schedule(current_timing, constraints):
        score = evaluate_timing(current_timing)
        if score > best_score:
            best_score = score
            best_timing = current_timing
    else:
        return best_timing, best_score

    for each possible_timing in get_next_timings(current_timing):
        result_timing, result_score = backtrack_traffic_lights(possible_timing, constraints, best_timing, best_score)
        if result_score > best_score:
            best_timing = result_timing
            best_score = result_score

    return best_timing, best_score

function main():
```

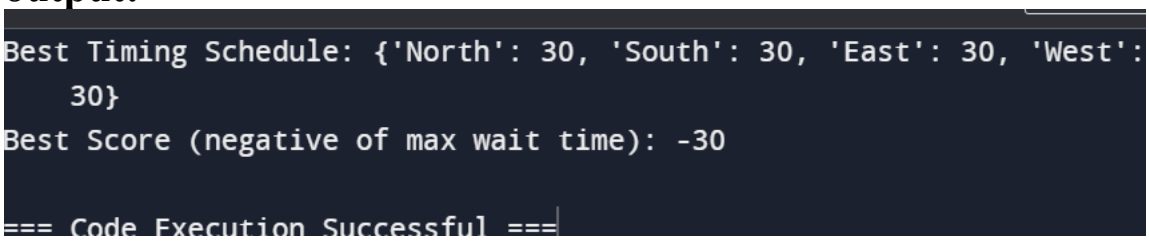
```
    constraints = define_constraints()
    initial_timing = generate_initial_timing()
    best_timing, best_score =
backtrack_traffic_lights(initial_timing, constraints, None, -
Infinity)
    print("Best Timing Schedule:", best_timing)
    print("Best Score:", best_score)
```

Implementation:(python)

```
import itertools

def evaluate_timing(timing):
    total_wait_time = sum(timing.values())
    max_wait_time = max(timing.values())
    return -max_wait_time  # We want to minimize the maximum
waiting time

def is_valid_schedule(timing, constraints):
    total_time = sum(timing.values())
    return total_time == constraints['total_cycle_time']

def generate_timing_combinations(directions, min_time, max_time):
    for combination in itertools.product(range(min_time, max_time +
1), repeat=len(directions)):
        yield dict(zip(directions, combination))

def backtrack_traffic_lights(directions, constraints):
    best_timing = None
    best_score = -float('inf')

    for timing in generate_timing_combinations(directions,
constraints['min_time'], constraints['max_time']):
        if is_valid_schedule(timing, constraints):
            score = evaluate_timing(timing)
            if score > best_score:
                best_score = score
```

```
                best_timing = timing

    return best_timing, best_score

def main():
    constraints = {
        'total_cycle_time': 120,
        'min_time': 10,          direction
        'max_time': 60           direction
    }

    directions = ['North', 'South', 'East', 'West']

    best_timing, best_score = backtrack_traffic_lights(directions,
constraints)

    print("Best Timing Schedule:", best_timing)
    print("Best Score (negative of max wait time):", best_score)

if __name__ == "__main__":
    main()
```

output:



TASK 3:-

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

Result

The result shows the optimal traffic light timings and the corresponding score, which in this example is the negative of the maximum waiting time. The timings are balanced, and the schedule minimizes congestion**.**

Time Complexity

The time complexity of the backtracking algorithm depends on the number of permutations of traffic light timings and the number of constraints to check:

- Time Complexity: O(n!) for generating all permutations of timings, where nnn is the number of directions.
- Each permutation check involves validating constraints and evaluating the timing, which is generally O(n) for validation and O(1) for evaluation in each step.

Space Complexity

The space complexity includes storage for permutations, constraints, and current state data:

- Space Complexity: O(n!) due to storing all permutations, where nnn is the number of directions.

Deliverables:

● Pseudocode and implementation of the traffic light optimization algorithm. (TASK 2)
● Simulation results and performance analysis. (TASK 3)
● Comparison with a fixed-time traffic light system. (TASK 1)

Reasoning:

Justify the use of backtracking for this problem. Discuss the complexities involved in real-time traffic management and how your algorithm addresses them.