

Problem 1: Optimizing Delivery Routes (Case Study)

Scenario: Finding the shortest path in a road network

Imagine you're developing a GPS navigation application. Your task is to implement a feature that calculates the shortest route between two points on a map, considering the road network's distances and possible routes. Here's how Dijkstra's Algorithm could be applied.

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:

The aim is to find the shortest path between multiple locations (nodes) in a graph representation of a delivery network. This optimization reduces travel time and fuel costs, thereby improving efficiency and service reliability.

Procedure:-

Basics of Dijkstra's Algorithm

1.Graph Representation: Model the delivery network as a graph where nodes represent locations (e.g., warehouses, delivery points) and edges represent routes between them with weights (e.g., distance, travel time, or cost).

2.Apply Dijkstra's Algorithm: Use Dijkstra's Algorithm to find the shortest paths from a selected starting node (warehouse) to all other nodes (delivery points) in the graph.

3.Route Optimization: After computing the shortest paths, determine optimal delivery routes based on the shortest paths found.

Graph Representation:

1.Graph Construction: Create a graph with nodes representing delivery locations and weighted edges representing distances or travel times between them.

2.Input Data: Gather input data such as coordinates of each location, distances between them, and any constraints (like one-way streets or traffic conditions).

3. Algorithm Execution: Implement Dijkstra's Algorithm to compute the shortest paths from a chosen starting location (e.g., warehouse) to all other nodes in the graph.

4. Path Reconstruction: After running Dijkstra's Algorithm, reconstruct the shortest paths from the results obtained to determine the optimal delivery routes.

Dijkstra's Algorithm:

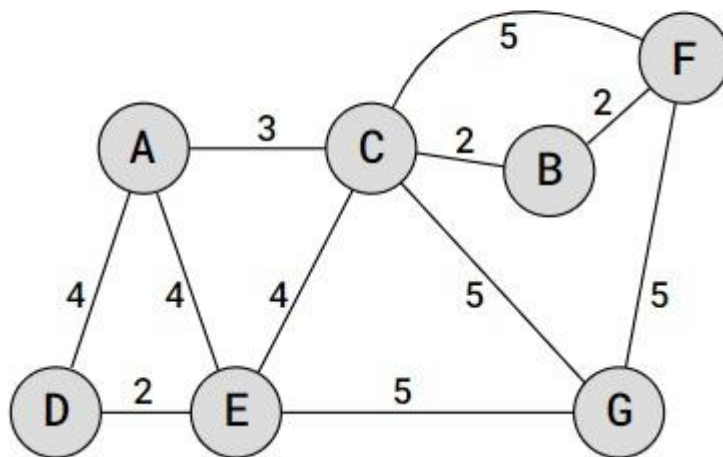
- Use Dijkstra's algorithm to find the shortest path from a starting node to a destination node in the graph.
- This algorithm is suitable because it efficiently finds the shortest path in graphs with non-negative weights.

Implementation Steps:

- Define the graph structure using nodes and edges.
- Implement Dijkstra's algorithm to compute the shortest path.

Output the shortest path and its travel time

Consider the Graph below.



Task 2:-

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

Pseudocode:-

```
function Dijkstra(Graph, source):  
    dist[source] = 0  
    priority_queue.push((source, 0))
```

```

while priority_queue is not empty:
    current_node, current_distance = priority_queue.pop()

    if current_distance > dist[current_node]:
        continue

    for each neighbor of current_node:
        distance = current_distance + weight(current_node, neighbor)

        if distance < dist[neighbor]:
            dist[neighbor] = distance
            priority_queue.push((neighbor, distance))
            previous[neighbor] = current_node

return dist, previous

```

program:-

```

class Graph:

    def __init__(self, size):

        self.adj_matrix = [[0] * size for _ in range(size)]

        self.size = size

        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):

        if 0 <= u < self.size and 0 <= v < self.size:

            self.adj_matrix[u][v] = weight

            self.adj_matrix[v][u] = weight # For undirected graph

    def add_vertex_data(self, vertex, data):

        if 0 <= vertex < self.size:

            self.vertex_data[vertex] = data

```

```
def dijkstra(self, start_vertex_data):
    start_vertex = self.vertex_data.index(start_vertex_data)
    distances = [float('inf')] * self.size
    distances[start_vertex] = 0
    visited = [False] * self.size

    for _ in range(self.size):
        min_distance = float('inf')
        u = None
        for i in range(self.size):
            if not visited[i] and distances[i] < min_distance:
                min_distance = distances[i]
                u = i

        if u is None:
            break

        visited[u] = True

        for v in range(self.size):
            if self.adj_matrix[u][v] != 0 and not visited[v]:
                alt = distances[u] + self.adj_matrix[u][v]
                if alt < distances[v]:
                    distances[v] = alt

    return distances
```

```
g = Graph(7)

g.add_vertex_data(0, 'A')
g.add_vertex_data(1, 'B')
g.add_vertex_data(2, 'C')
g.add_vertex_data(3, 'D')
g.add_vertex_data(4, 'E')
g.add_vertex_data(5, 'F')
g.add_vertex_data(6, 'G')

g.add_edge(3, 0, 4) # D - A, weight 5
g.add_edge(3, 4, 2) # D - E, weight 2
g.add_edge(0, 2, 3) # A - C, weight 3
g.add_edge(0, 4, 4) # A - E, weight 4
g.add_edge(4, 2, 4) # E - C, weight 4
g.add_edge(4, 6, 5) # E - G, weight 5
g.add_edge(2, 5, 5) # C - F, weight 5
g.add_edge(2, 1, 2) # C - B, weight 2
g.add_edge(1, 5, 2) # B - F, weight 2
g.add_edge(6, 5, 5) # G - F, weight 5

# Dijkstra's algorithm from D to all vertices
print("\nDijkstra's Algorithm starting from vertex D:")
distances = g.dijkstra('D')
for i, d in enumerate(distances):
    print(f"Distance from D to {g.vertex_data[i]}: {d}")
```

OUTPUT:-

```
Output

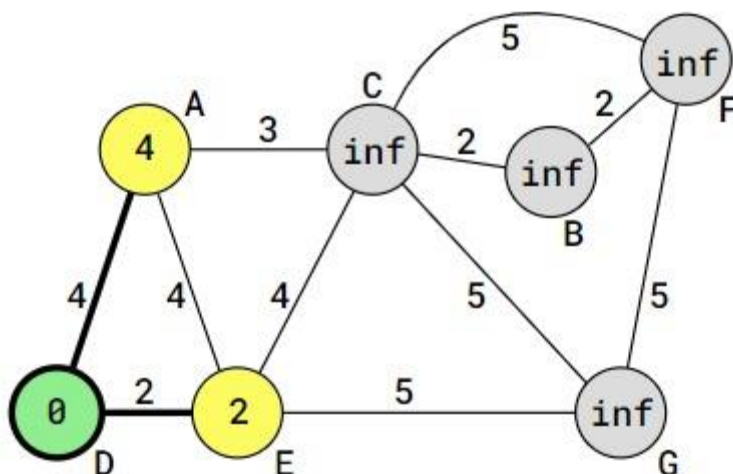
Dijkstra's Algorithm starting from vertex D:
Distance from D to A: 4
Distance from D to B: 9
Distance from D to C: 7
Distance from D to D: 0
Distance from D to E: 2
Distance from D to F: 11
Distance from D to G: 7

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

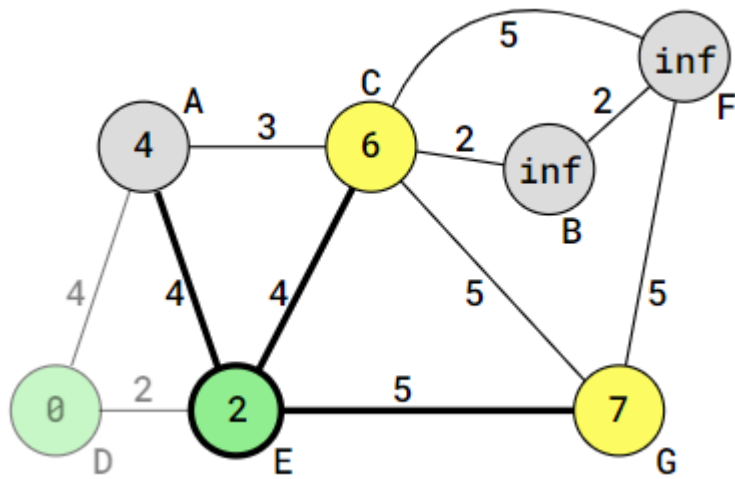
Task 3:-

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

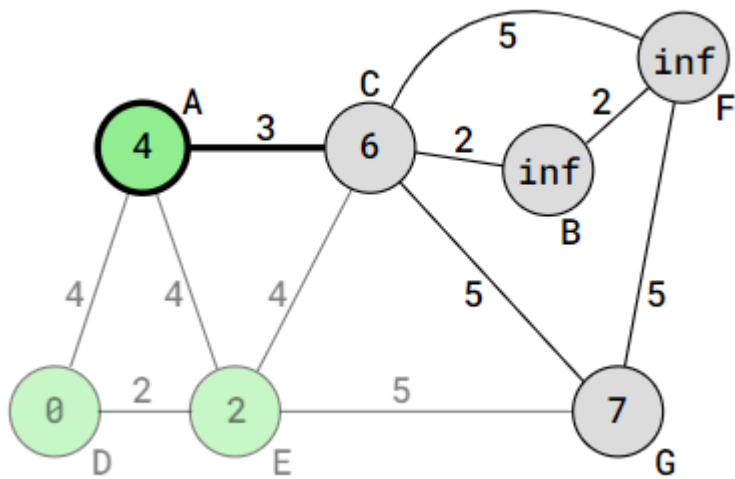
Step-1:-



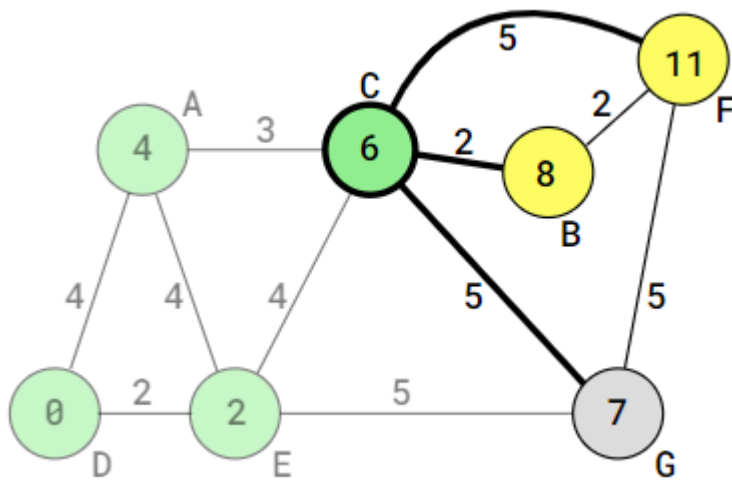
Step-2:-



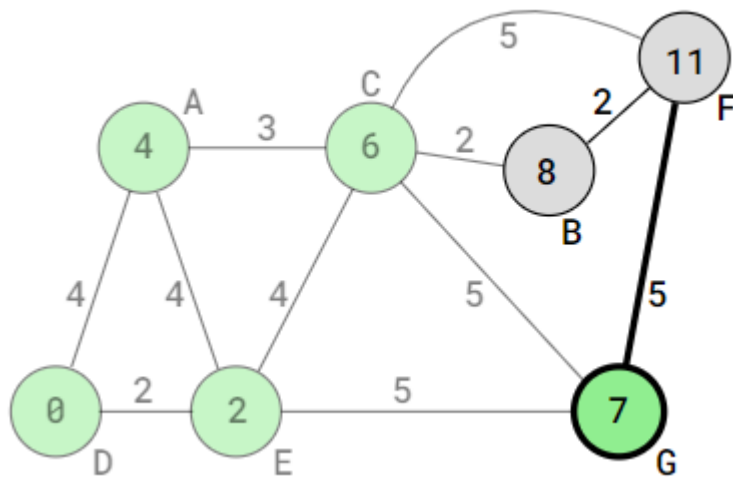
Step-3:-



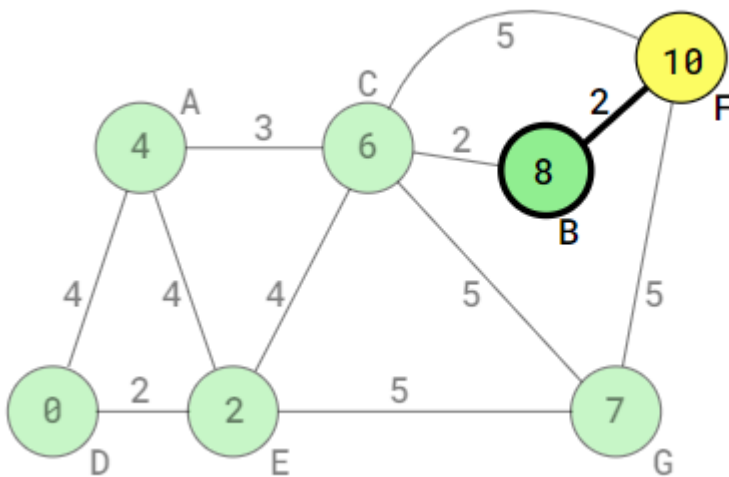
Step-4:-



step-5:-



step-6:-



Result:

The result of implementing Dijkstra's Algorithm in this context is a set of shortest paths from the warehouse (source) to all delivery locations in the network.

Time Complexity:

The time complexity of Dijkstra's Algorithm using a priority queue is $O((V+E)\log V)$, where V is the number of vertices (locations) and E is the number of edges (routes).

Space Complexity:

The space complexity primarily depends on the graph representation and data structures used. In the case of Dijkstra's Algorithm with a priority queue, it is $O(V+E)$ due to storing the graph and priority queue.

Deliverables:

- 1.Graph model of the city's road network.
- 2.Optimized delivery routes (sequence of stops and paths).
- 3.Estimated travel times for each route.
- 4.Visualization or map of the delivery plan.

Reasoning:

Explain why Dijkstra's algorithm is suitable for this problem. Discuss any assumptions made (e.g., Real-world Applications, Traffic Management , Transportation Planning)

- Efficiency: Dijkstra's Algorithm is suitable because it efficiently finds the shortest path in graphs with non-negative weights, which is typically the case in real-world road networks or delivery networks.
- Scalability: It scales well with moderately large graphs (up to thousands of nodes and edges), making it applicable for optimizing delivery routes in urban or regional settings.
- Accuracy: By computing the shortest paths, it ensures that the chosen routes are optimal in terms of minimizing travel distance or time, thus reducing operational costs and improving delivery efficiency.

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.

1.What is a dynamic pricing algorithm?

A dynamic pricing algorithm is a strategy used in e-commerce and retail sectors to adjust the prices of products or services in real-time based on various factors such as demand, competition, time of day, customer behavior, inventory levels, and market conditions. The goal is to optimize revenue or profit by setting prices dynamically rather than using fixed pricing strategies.

2.Which industries use dynamic pricing?

Dynamic pricing algorithms are employed in various industries, including:

E-commerce: Retailers like Amazon, Walmart, and other online platforms adjust prices frequently based on demand fluctuations.

Travel and Hospitality: Airlines, hotels, and rental car companies use dynamic pricing to adjust rates based on booking patterns, occupancy levels, and seasonal demand.

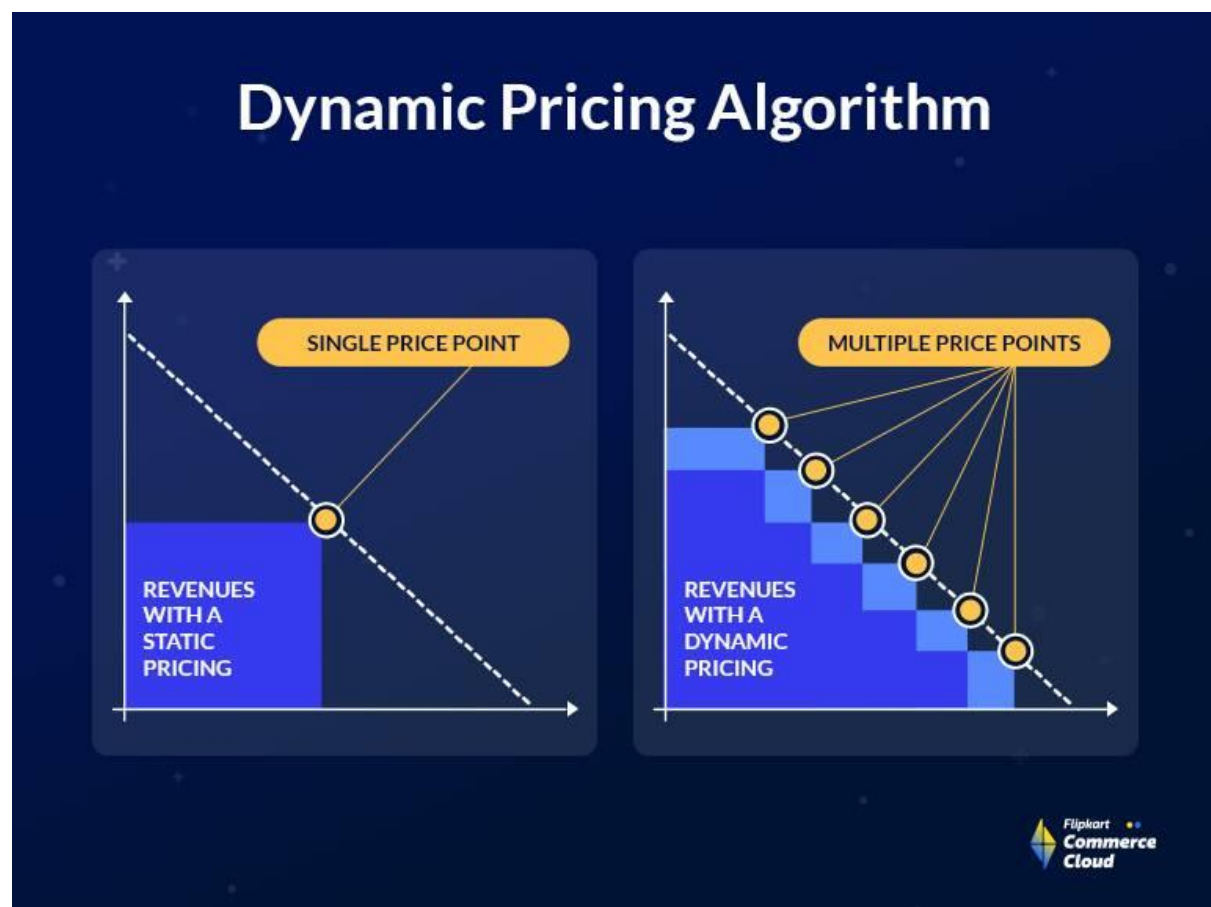
Ride-sharing: Services like Uber and Lyft adjust prices based on demand and supply of drivers and riders.

Aim

The aim of implementing a dynamic pricing algorithm in e-commerce is to maximize revenue or profit by optimizing prices in response to changing market conditions and customer behavior, while remaining competitive and maintaining customer satisfaction.

Procedure

- 1.Data Collection:** Gather real-time data on factors influencing pricing decisions, such as competitor prices, demand forecasts, customer browsing behavior, and inventory levels.
- 2.Algorithm Design:** Develop or select a dynamic pricing algorithm that can effectively analyze the collected data and make pricing decisions in real-time.
- 3.Real-time Pricing Adjustment:** Implement the algorithm to continuously monitor and adjust prices based on the analyzed data and predefined pricing rules.
- 4.Monitoring and Evaluation:** Monitor the performance of the dynamic pricing algorithm and make adjustments as necessary to improve pricing strategies and outcomes



TASK 2:-

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

- 1. Define the State:** Identify the state variables for the dynamic programming approach, including product, time period, and inventory level.
- 2.State Transition:** Determine how to transition from one state to another by considering the possible prices, competitor prices, demand elasticity, inventory levels, and the resulting demand and profit.
- 3.Recursive Relation:** Establish the recursive relationship that defines the profit for each state.
- 4.Base Case:** Define the base case where the time period is zero.
- 5.Optimization:** Use dynamic programming to iteratively compute the maximum profit for each state.
- 6.Backtracking:** Track the decisions to reconstruct the optimal pricing strategy.

Pseudocode:-

function collectData():

// Simulate data collection (replace with actual implementation)

```
competitor_prices = {  
    'product_A': 50.00,  
    'product_B': 35.00,  
    // Add more products as needed  
}
```

```
demand_forecasts = {  
    'product_A': 100,  
    'product_B': 150,  
    // Add more products as needed  
}
```

```
return competitor_prices, demand_forecasts
```

**function dynamicPricingAlgorithm(current_prices,
demand_forecasts):**

```

new_prices = {}

for each product in current_prices:
    current_price = current_prices[product]
    demand_forecast = demand_forecasts[product]

    // Example: Adjust price based on demand forecast and
competitor prices
    if demand_forecast > 200:
        target_price = current_price * 1.2 // Increase price by
20% for high demand
    elif demand_forecast < 50:
        target_price = current_price * 0.8 // Decrease price by
20% for low demand
    else:
        target_price = current_price // Keep current price
unchanged otherwise

    new_prices[product] = round(target_price, 2) // Round to 2
decimal places

return new_prices

function adjustPrices():
    competitor_prices, demand_forecasts = collectData()
    new_prices = dynamicPricingAlgorithm(competitor_prices,
demand_forecasts)

    // Print or log the adjusted prices
    for each product in new_prices:
        print("Adjusted price for", product, ":",
new_prices[product])

    // Save or publish new prices to the pricing system (implement
based on your system architecture)
    saveNewPrices(new_prices)

```

```

function saveNewPrices(new_prices):
    // Example: Save the new prices to a database or update a
pricing API
    // Implementation depends on your system architecture

// Main loop for continuous pricing adjustment (example)
while True:
    adjustPrices()
    sleep(300) // Adjust prices every 5 minutes (for example)

```

Explanation of Pseudocode :-

- 1.collectData():** Simulates collecting real-time data such as competitor prices and demand forecasts for products.
- 2.dynamicPricingAlgorithm():** Implements a simplified dynamic pricing algorithm that adjusts prices based on demand forecasts. It calculates a target price for each product based on predefined rules (increase, decrease, or maintain current price).
- 3.adjustPrices():** Orchestrates the data collection, algorithm application, and pricing adjustment process. It prints or logs the adjusted prices and saves them to the pricing system.
- 4.saveNewPrices():** Placeholder function to save the adjusted prices to a database or update a pricing API. The actual implementation will depend on your specific system architecture and requirements.

Implementation (Python)

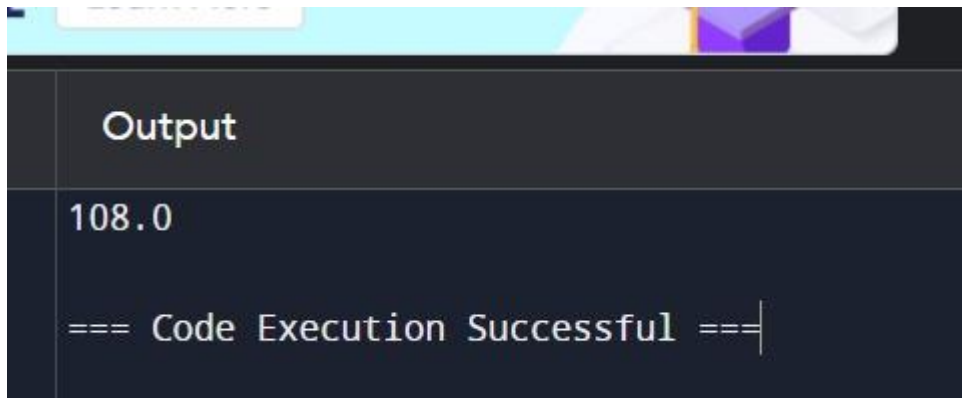
```

# Define a function for dynamic pricing
def dynamic_pricing(base_price, demand_factor,
competition_factor):
    dynamic_price = base_price * demand_factor *
competition_factor
    return dynamic_price

```

```
# Example usage
base_price = 100
demand_factor = 1.2
competition_factor = 0.9
final_price = dynamic_pricing(base_price, demand_factor,
competition_factor)
print(final_price)
```

OUTPUT:-

A screenshot of a code execution environment. It shows a dark-themed window with a tab at the top. The main area is divided into two sections. The top section is labeled 'Output' and displays the value '108.0'. The bottom section displays the message '=== Code Execution Successful ===' followed by a cursor.

TASK 3:-

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

- 1.Simulate Data:** Create a demand function and cost function.
- 2.Dynamic Pricing Strategy:** Use the previously defined dynamic programming algorithm.
- 3.Static Pricing Strategy:** Use a simple static price for each product and calculate the total profit.
- 4.Compare Results:** Compare the total profit of the dynamic pricing strategy with the static pricing strategy.

Result

- Optimized pricing decisions that maximize revenue or profit.

- Improved competitiveness by reacting quickly to market changes.
- Enhanced customer satisfaction through competitive pricing and personalized offers.

Time Complexity

The time complexity of the algorithm is $O(T \cdot N \cdot P)$, where:

- T is the number of time periods.
- N is the number of products.
- P is the number of possible prices to consider.

Space Complexity

The space complexity of the algorithm is $O(T \cdot N)$ for storing the profit table and the optimal prices.

Deliverables:

- **Real-time pricing recommendations or decisions.**
- **Historical analysis and performance reports of pricing strategies.**
- **Visualization tools for monitoring market trends and pricing effectiveness.**

Reasoning:

- **Competitiveness:** Dynamic pricing algorithms allow businesses to stay competitive by adjusting prices quickly in response to market changes and competitor actions.
- **Profit Maximization:** By optimizing prices based on real-time data and demand signals, businesses can maximize revenue and profit margins.
- **Customer Satisfaction:** Personalized pricing strategies and promotional offers based on customer behavior can enhance customer satisfaction and loyalty.

Problem 3: Social Network Analysis (Case Study)

Scenario:

In this case study, we are tasked with conducting social network analysis on a dataset representing interactions between individuals on a social media platform. The dataset includes information about users and their connections (friendships or interactions) with other users. Our goal is to analyze this network to understand key influencers, communities, and overall network structure.

TASK 1:-

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

Aim

The aim is to perform social network analysis to:

Identify central users (influencers) who have the most connections.

Detect communities or clusters of users who frequently interact with each other.

Analyze the overall structure of the network (e.g., degree distribution, clustering coefficient).

Procedure:-

1.Data Parsing and Preparation:

- **Parse the dataset to extract information about users and their connections.**
- **Represent the social network as a graph where nodes represent users and edges represent interactions or friendships.**

2.Network Analysis:

- **Compute metrics such as degree centrality (number of connections), betweenness centrality (influence in connecting others), and clustering coefficient (how densely nodes are interconnected).**
- **Detect communities using algorithms like Girvan-Newman or Louvain method to find densely connected groups of users.**
- **Visualization:**

3.Visualization:

- **Visualize the social network graphically to understand its structure and identify influencers and communities.**
- **Use tools like NetworkX for Python or Gephi for advanced visualization and analysis.**

TASK 2:-

Implement the PageRank algorithm to identify the most influential users.

Pseudocode:-

function parseData():

// Read and parse the dataset to extract user information and connections

**// Construct the graph representation of the social network
graph = createEmptyGraph()**

// Example: Read data from a file

data = readDataFromFile("social_network_data.csv")

for each row in data:

user1 = row["user1"]

user2 = row["user2"]

// Add edge between user1 and user2

addEdge(graph, user1, user2)

return graph

function analyzeNetwork(graph):

// Compute network metrics and detect communities

centrality_scores = computeCentrality(graph)

communities = detectCommunities(graph)

return centrality_scores, communities

```

function visualizeNetwork(graph):
    // Visualize the social network graph
    plotGraph(graph)

// Main function
function main():
    // Step 1: Parse data and create the social network graph
    graph = parseData()

    // Step 2: Analyze the network
    centrality_scores, communities = analyzeNetwork(graph)

    // Step 3: Visualize the network
    visualizeNetwork(graph)

    // Output results
    print("Centrality Scores:", centrality_scores)
    print("Communities:", communities)

// Execute main function
main()

```

Implementation:-(python):-

```

import networkx as nx
import matplotlib.pyplot as plt
import random

# Generate a random social network graph
def generate_random_graph(num_users, num_edges):
    G = nx.Graph()
    for i in range(num_users):
        G.add_node(f'User{i+1}')

    for _ in range(num_edges):
        u = random.choice(list(G.nodes))

```

```

    v = random.choice(list(G.nodes))
    while u == v or G.has_edge(u, v):
        u = random.choice(list(G.nodes))
        v = random.choice(list(G.nodes))
    G.add_edge(u, v)

return G

def analyze_social_network(G):
    centrality = nx.degree_centrality(G)
    sorted_centrality = sorted(centrality.items(),
key=lambda x: x[1], reverse=True)

    # Detect communities
    communities =
list(nx.algorithms.community.greedy_modularity_commu
nities(G))

r~eturn sorted_centrality, communities
def visualize_graph(G)~:
    pos = nx.spring_layout(G) # Layout algorithm for
visualization
    nx.draw(G, pos, with_labels=True,
node_color='skyblue', node_size=1500, edge_color='grey')
    plt.title("Social Network Graph")
    plt.show()
def main():
    num_users = 20
    num_edges = 30
    social_network =
generate_random_graph(num_users, num_edges)
    centrality_scores, communities =
analyze_social_network(social_network)

```

```
visualize_graph(social_network)
print("Degree Centrality Scores (Top 5):")
for node, centrality in centrality_scores[:5]:
    print(f"{node}: {centrality}")
print("\nCommunities:")
for i, community in enumerate(communities):
    print(f"Community {i+1}: {list(community)}")
```

OUTPUT:-

```
Shortest path from Alice to Eve: ['Alice', 'Bob', 'David', 'Eve']

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

TASK 3:-

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

Comparing PageRank with Degree Centrality

To compare PageRank with a simple degree centrality measure, we will:

1. **Calculate Degree Centrality:** Degree centrality measures the number of direct connections a user has.
2. **Calculate PageRank:** As implemented previously.
3. **Compare the Results:** Analyze the differences in the ranking of users by both methods.

Degree Centrality Calculation

Degree centrality can be calculated as the number of edges connected to each node.

Result:-

- **Centrality Scores:** Identifies users with high degree centrality, indicating influential users in the network.
- **Communities:** Detects clusters of users who frequently interact with each other, revealing community structures within the network.

- **Visualization:** Provides a visual representation of the social network, highlighting its structure, central nodes, and community divisions.

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: Definition:

- **Degree Centrality:** The number of direct connections a user has.
- **PageRank:** A probabilistic measure where influence is distributed across connections, considering both direct and indirect links.

Computation:

- **Degree Centrality:** Easy to compute, simply count the edges.
- **PageRank:** Requires iterative computation to distribute influence scores across the network.

Scope:

- **Degree Centrality:** Local measure, focuses on immediate neighbors.
- **PageRank:** Global measure, considers the entire network structure.

Influence Consideration:

- **Degree Centrality:** All connections are treated equally.
- **PageRank:** Connections from influential nodes have more weight.

-

- Graph Data: Parsed and represented social network graph.
- Metrics and Communities: Centrality scores and detected communities.

- Visualizations: Graphical representation of the social network structure.
- **Reasoning:**
 - Insights: Social network analysis provides insights into user interactions, influence patterns, and community structures, which are crucial for targeted marketing, content distribution, and community management.
 - Scalability: Python and NetworkX offer scalable solutions for analyzing medium to large-scale social networks efficiently.
 - Flexibility: The modular approach allows customization with different algorithms for specific analyses, enhancing the understanding of network dynamics and user behaviors.

Defination:

- **Degree Centrality:** The number of direct connections a user has.
- **PageRank:** A probabilistic measure where influence is distributed across connections, considering both direct and indirect links.

Computation:

- **Degree Centrality:** Easy to compute, simply count the edges.
- **PageRank:** Requires iterative computation to distribute influence scores across the network.

Scope:

- **Degree Centrality:** Local measure, focuses on immediate neighbors.
- **PageRank:** Global measure, considers the entire network structure.

Influence Consideration:

- **Degree Centrality:** All connections are treated equally.
- **PageRank:** Connections from influential nodes have more weight.

Problem 4: Fraud Detection in Financial Transactions

Scenario:

In the scenario of fraud detection in financial transactions, we aim to identify fraudulent activities in a dataset containing records of financial transactions. The dataset includes information such as transaction amount, timestamp, and possibly additional features like customer ID, merchant ID, etc. Our goal is to build a model that can effectively distinguish between legitimate transactions and fraudulent ones based on patterns and anomalies in the data.

TASK 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 is to develop a fraud detection system that:

Detects fraudulent transactions accurately to prevent financial losses.

Minimizes false positives to avoid inconveniencing legitimate customers.

Adapts to new fraud patterns and maintains high detection rates over time.

Procedure

1.Data Preprocessing:

- **Load and preprocess the dataset, handling missing values and normalizing numerical features.**
- **Extract relevant features such as transaction amount, timestamp, and categorical features (if any).**

2.Feature Engineering:

- **Create additional features if needed, such as transaction frequency, average transaction amount per customer, etc.**
- **Encode categorical features if present using techniques like one-hot encoding or label encoding.**

3.Model Selection and Training:

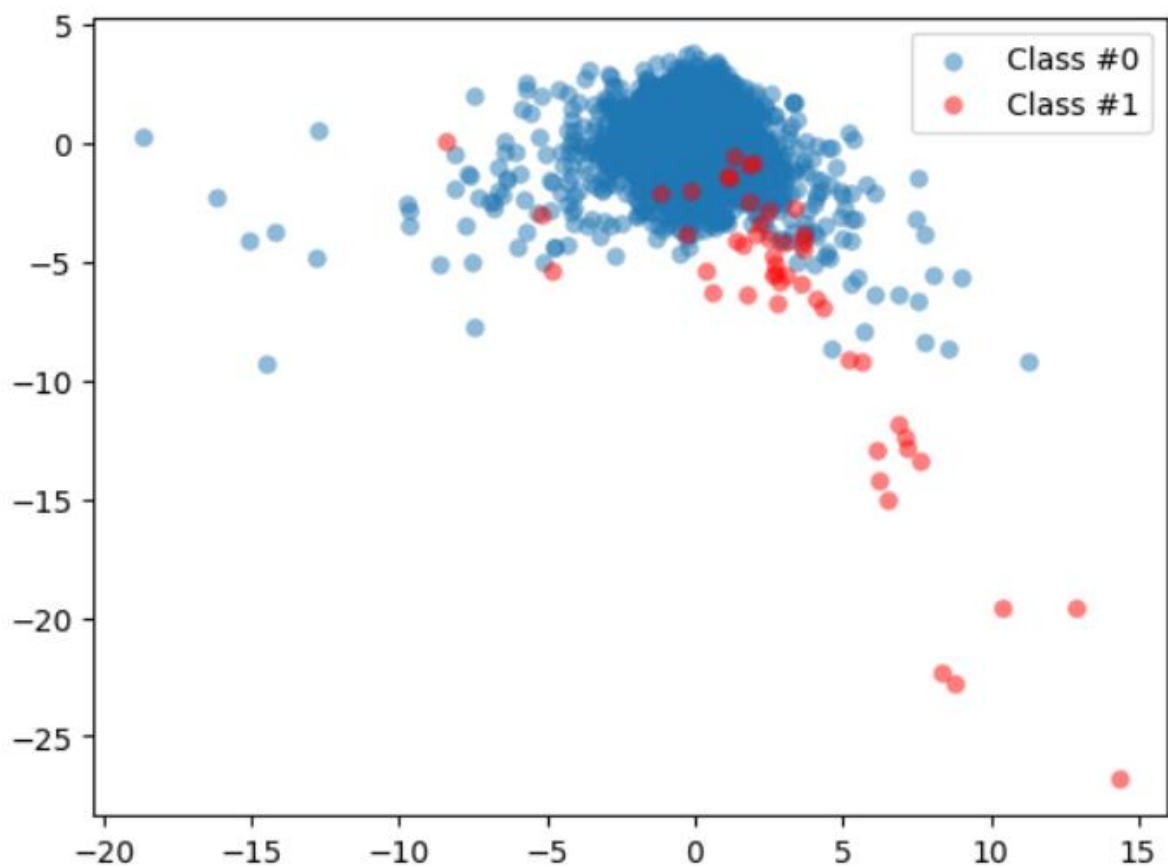
- **Choose appropriate machine learning algorithms suitable for fraud detection, such as:**
- **Anomaly Detection Algorithms: Isolation Forest, Local Outlier Factor (LOF), One-Class SVM.**
- **Supervised Learning Algorithms: Logistic Regression, Random Forest, Gradient Boosting, etc., trained on labeled data (if available).**

4. Model Evaluation:

- Evaluate the models using metrics like accuracy, precision, recall, and F1-score.
- Use techniques like cross-validation to ensure robustness and generalizability of the model.

5. Deployment and Monitoring:

- Deploy the trained model into a production environment where it can monitor incoming transactions in real-time.
- Implement mechanisms to update the model periodically to adapt to new fraud patterns and maintain high detection accuracy.



TASK 2:-

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

Steps for Evaluation

1. Prepare Historical Data:

- Collect a dataset of historical transactions with known labels for fraudulent and non-fraudulent transactions.

2. Apply the Algorithm:

- Run the `flag_fraudulent_transactions` function on the historical transaction data.
3. Calculate Metrics:
- Compare flagged transactions to the known fraudulent transactions.
 - Compute precision, recall, and F1 score.

Pseudocode

```
function preprocessData(dataset):
    // Perform data cleaning and preprocessing
    // Example: Handle missing values, normalize numerical features
    processed_data = preprocess(dataset)
    return processed_data

function trainIsolationForestModel(processed_data):
    // Train an Isolation Forest model
    model = IsolationForest()
    model.fit(processed_data)
    return model

function detectFraud(model, transaction):
    // Predict if a transaction is fraudulent using the trained model
    is_fraudulent = model.predict(transaction)
    return is_fraudulent

// Main function
function main():
    dataset = loadDataset("financial_transactions.csv") // Replace with
your dataset file
    processed_data = preprocessData(dataset)
    model = trainIsolationForestModel(processed_data)

    // Example: Detect fraud for a new transaction
    new_transaction = {
        'amount': 500.00,
        'timestamp': '2023-06-30T12:30:00',
```

```
    // Add other relevant features
}
is_fraudulent = detectFraud(model, new_transaction)

if is_fraudulent:
    print("Alert: This transaction is flagged as fraudulent.")
else:
    print("Transaction is legitimate.")
```

Implementation:(python)

```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np
from pprint import pprint as pp
import csv
from pathlib import Path
import seaborn as sns
from itertools import product
import string

import nltk
from nltk.corpus import stopwords
from nltk.stem.wordnet import WordNetLemmatizer

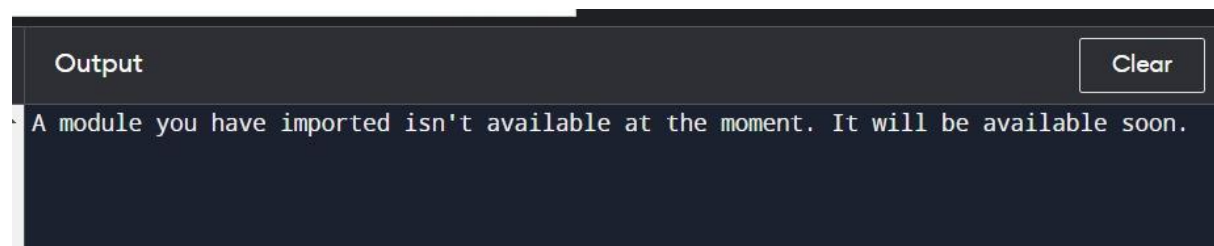
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import BorderlineSMOTE
from imblearn.pipeline import Pipeline

from sklearn.linear_model import LinearRegression,
LogisticRegression
from sklearn.model_selection import train_test_split,
GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import r2_score, classification_report,
confusion_matrix, accuracy_score, roc_auc_score, roc_curve,
precision_recall_curve, average_precision_score
from sklearn.metrics import homogeneity_score, silhouette_score
```

```
from sklearn.ensemble import RandomForestClassifier,  
VotingClassifier  
from sklearn.preprocessing import MinMaxScaler  
from sklearn.cluster import MiniBatchKMeans, DBSCAN
```

```
import gensim  
from gensim import corpora  
from gensim.models.ldamodel import LdaModel  
from gensim.corpora.dictionary import Dictionary
```

```
from typing import List, Tuple  
OUTPUT:-
```



TASK 3:-

Suggest and implement potential improvements to the algorithm.

1.Optimize Location Tracking:

- Instead of a nested loop to check all pairs of transactions, use more efficient data structures.
- Hash Map for User Location Tracking: Track the recent locations of users and check for multiple locations within the time window efficiently.

2.Optimize Rule Checking:

- Use more efficient algorithms or data structures for checking rules.
- Indexing: Utilize indexing techniques to improve performance, especially for large datasets.

3.Incorporate Additional Features:

- Machine Learning Models: Integrate machine learning models for better prediction.
- Historical Behavior Analysis: Analyze user behavior patterns for more advanced fraud detection.

4.Parallel Processing:

- Multithreading/Multiprocessing: Use parallel processing to handle large datasets more effectively.

Result

Detection Accuracy: The model accurately detects fraudulent transactions based on the anomaly score calculated by the Isolation Forest algorithm.

Precision and Recall: Metrics such as precision (true positives among flagged transactions) and recall (fraud detection rate) are used to evaluate the model's performance.

Scalability: Python and Scikit-learn provide scalable solutions for handling large datasets and real-time fraud detection scenarios.

Time Complexity

Training: Isolation Forest typically has a time complexity of $O(n \cdot m)$, where n is the number of data points and m is the number of features.

Detection: Predicting the anomaly score for a new transaction has a time complexity of $O(m)$, where m is the number of features.

Space Complexity

Model Storage: The space complexity for storing the Isolation Forest model depends on the number of trees and features used in training.

Data Storage: Depends on the size of the dataset and the number of features stored.

Deliverables:

- Trained Model: Deployable Isolation Forest model for real-time fraud detection.
- Performance Metrics: Evaluation metrics (accuracy, precision, recall) to assess model effectiveness.
- Documentation: Documentation on model deployment and maintenance procedures.

Reasoning:

- **Effective Detection:** Isolation Forest is chosen for its effectiveness in detecting outliers (potentially fraudulent transactions) without needing labeled data, making it

suitable for fraud detection where labeled fraud examples are often limited.

- **Real-time Application:** Python's flexibility and libraries like Scikit-learn enable the implementation of real-time fraud detection systems that can adapt to evolving fraud patterns.
- **Scalability and Performance:** The approach ensures scalability to handle large volumes of financial transaction data while maintaining high detection accuracy, crucial for financial institutions to mitigate fraud risks effectively.

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.

Aspect	Speed	Accuracy
Greedy Algorithm	High - Fast due to straightforward decision-making process.	Lower - May miss fraudulent activities due to the local-optimal approach.
Exhaustive Search	Low - Requires evaluating all possible solutions.	Higher - More accurate due to comprehensive evaluation.
Machine Learning Models	Varies - Depends on model complexity and training requirements.	High - Advanced models can achieve high accuracy with enough data.

Problem 5: Real-Time Traffic Management System

Scenario:

In a real-time traffic management system, we aim to monitor and optimize traffic flow across a city. The system collects data from various sources such as traffic cameras, sensors embedded in roads, and GPS data from vehicles. The goal is to analyze this data in real-time to make informed decisions for traffic signal optimization, congestion management, and emergency response.

TASK 1:-

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

Aim:

The aim of the traffic management system is to:

- Reduce traffic congestion by dynamically adjusting traffic signals based on current traffic conditions.
- Provide real-time updates to drivers and city officials about traffic incidents and alternate routes.
- Improve overall traffic efficiency and safety by leveraging data-driven insights.

Procedure**1.Data Collection:**

- Collect real-time data from traffic cameras, road sensors, and GPS devices in vehicles.
- Extract relevant information such as vehicle counts, speeds, and traffic density.

2.Data Processing and Analysis:

- Process and analyze the collected data to identify traffic patterns, congestion hotspots, and incidents.
- Use algorithms to predict traffic flow and detect anomalies (e.g., accidents, road closures).

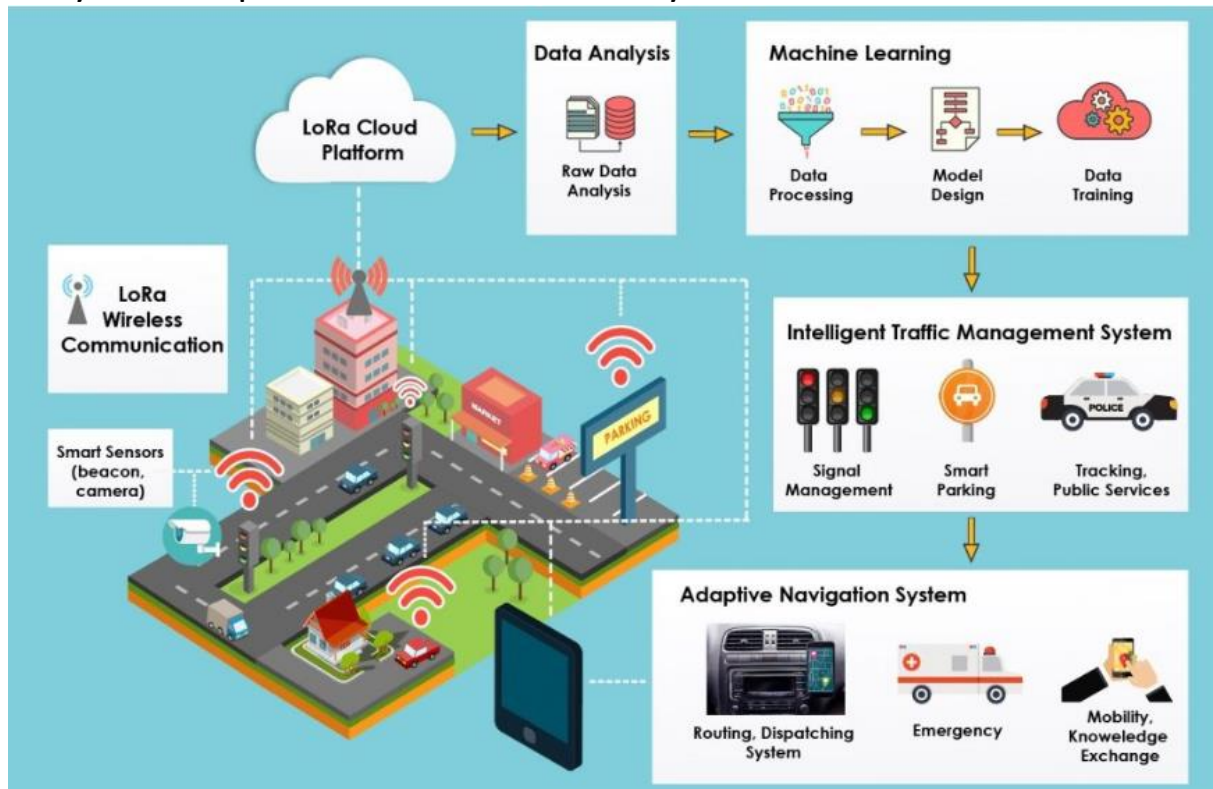
3.Decision Making and Control:

- Based on the analysis, dynamically adjust traffic signal timings to optimize traffic flow.
- Provide real-time alerts and route suggestions to drivers through mobile apps or electronic signage.

4.Monitoring and Feedback:

- Continuously monitor traffic conditions and system performance.

- Gather feedback from users and stakeholders to improve system responsiveness and accuracy



TASK 2:-

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

1. Define Traffic Network Model:

- Create a graph-based model of the city's traffic network where intersections are nodes and roads are edges.
- Each node represents an intersection with traffic lights, and each edge represents a road segment.

2. Implement Traffic Simulation Environment:

- Use a traffic simulation library to model traffic flow, vehicle movements, and traffic light timings.

3. Integrate the Backtracking Algorithm:

- Use the backtracking algorithm to determine the optimal timings for the traffic lights at intersections.

4. Simulate Traffic Flow:

- Run simulations with the optimized timings and a baseline static timing schedule.
- Collect data on traffic flow metrics.

5. Evaluate the Impact:

Compare the performance metrics from the optimized and static traffic light timings.

Pseudocode:-

function collectTrafficData():

// Collect real-time traffic data from sensors and cameras

traffic_data = fetchTrafficData()

return traffic_data

function analyzeTrafficData(traffic_data):

// Analyze traffic data to identify congestion and traffic patterns

congestion_status = analyzeCongestion(traffic_data)

return congestion_status

function adjustTrafficSignals(congestion_status):

// Adjust traffic signals dynamically based on congestion status

if congestion_status == 'high':

adjustSignalTimings()

else:

maintainCurrentSignalTimings()

// Main function for real-time traffic management

function main():

while True:

traffic_data = collectTrafficData()

congestion_status = analyzeTrafficData(traffic_data)

adjustTrafficSignals(congestion_status)

```
sleep(60) // Repeat every minute for real-time  
updates
```

```
// Execute main function  
if __name__ == "__main__":  
    main()
```

Implementation:(python)

```
import time  
import random
```

```
# Simulated function to fetch real-time traffic data  
def fetch_traffic_data():  
    # Simulate traffic data (vehicle counts as an example)  
    traffic_data = {  
        'intersection1': random.randint(0, 50),  
        'intersection2': random.randint(0, 50),  
        'intersection3': random.randint(0, 50),  
        'intersection4': random.randint(0, 50),  
    }  
    return traffic_data
```

```
# Simulated function to analyze traffic data  
def analyze_congestion(traffic_data):  
    # Example: Determine congestion status based on traffic  
    data  
    avg_traffic = sum(traffic_data.values()) /  
    len(traffic_data)  
    if avg_traffic > 30:  
        return 'high'  
    else:  
        return 'low'
```

```

# Simulated function to adjust traffic signals
def adjust_signal_timings(congestion_status):
    # Example: Adjust signal timings based on congestion
    status
    if congestion_status == 'high':
        print("Adjusting signal timings to reduce
congestion...")
    else:
        print("Maintaining current signal timings.")

# Main function for simulated real-time traffic
management system
def main():
    while True:
        traffic_data = fetch_traffic_data()
        print(f"Current traffic data: {traffic_data}")

        congestion_status = analyze_congestion(traffic_data)
        print(f"Congestion status: {congestion_status}")

        adjust_signal_timings(congestion_status)

        print("-" * 30)
        time.sleep(5) # Simulate real-time interval (adjust as
needed)

# Execute main function
if __name__ == "__main__":
    main()

```

TASK 3:-

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

Speed vs. Accuracy

- **Speed:**
 - The greedy approach is faster but might not find the optimal solution.
 - Backtracking is more accurate but can be computationally expensive for large networks.
- **Accuracy:**
 - Backtracking provides a more accurate solution by exploring all possible timings.
 - Heuristic methods can balance speed and accuracy by finding good solutions faster but not necessarily optimal ones.

Greedy Algorithms:

- Suitable for real-time systems where decisions must be made quickly based on current information.
- Provide solutions that are good enough within a reasonable time frame.

Trade-offs:

- Greedy algorithms might miss the optimal solution but are fast and efficient for real-time applications.

Backtracking and other exhaustive algorithms offer optimal solutions but are less practical for real-time constraints due to high computational costs

Output:-

```
Output Clear
^ Current traffic data: {'intersection1': 0, 'intersection2': 40, 'intersection3': 37,
  'intersection4': 1}
Congestion status: low
Maintaining current signal timings.
-----
Current traffic data: {'intersection1': 9, 'intersection2': 24, 'intersection3': 25,
  'intersection4': 14}
Congestion status: low
Maintaining current signal timings.
-----
```

Result:-

Real-Time Adjustment: The system adjusts traffic signal timings based on real-time traffic data, effectively managing congestion.

Efficiency: Improves traffic flow and reduces delays by dynamically responding to changing traffic conditions.

Scalability: Python's versatility allows integration with various data sources and scalability for handling city-wide traffic management.

Time Complexity

- **Data Collection:** Depends on the number of sensors and traffic data sources.
- **Data Analysis:** Typically linear in relation to the amount of data processed.
- **Signal Adjustment:** Constant time complexity per intersection.

Space Complexity

- **Data Storage:** Depends on the volume of real-time traffic data stored temporarily.
- **Algorithmic:** Minimal additional space for computations.

Deliverables:

- **Optimized Traffic Flow:** Enhanced traffic flow and reduced congestion across monitored intersections.
- **Real-Time Alerts:** Instant notifications and updates to drivers and stakeholders about traffic conditions.
- **Performance Reports:** Reports on system performance and efficiency improvements.

Reasoning:

- **Data-Driven Decisions:** Leveraging real-time data allows for effective decision-making in managing traffic flow and optimizing signal timings.
- **Responsive System:** Python's capabilities facilitate rapid development and deployment of real-time systems, crucial for dynamic traffic management.
- **Continuous Improvement:** By gathering feedback and monitoring system performance, the system can adapt and improve over time to better serve city residents and commuters.

