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:

The aim is to model and analyse a city's road network as a graph, where intersections are nodes and roads are edges with weights representing travel time. We will then demonstrate how to find the shortest path between two nodes using Dijkstra's algorithm.

Procedure:

Basics of Dijkstra's Algorithm:

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.

- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.

- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.

- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

  Graph Representation:

  - Represent the city's road network as a weighted graph.
  - Nodes represent intersections.
  - Edges represent roads between intersections, with weights representing travel times.

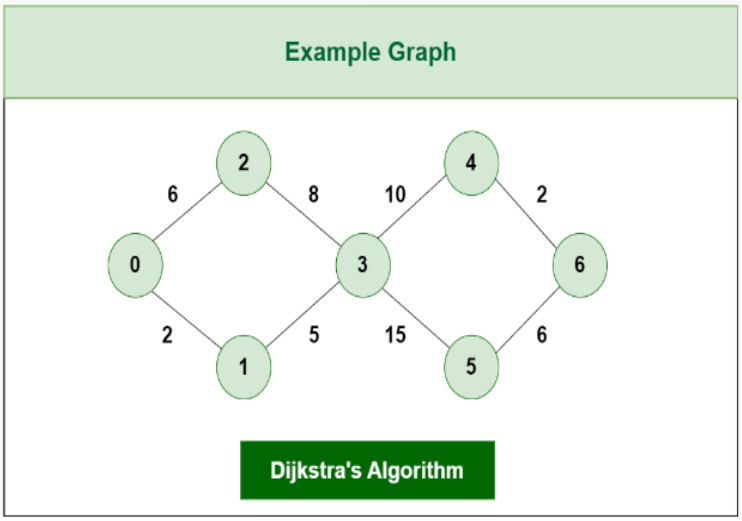
  Dijkstra's Algorithm:

  - 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.



Task 2:

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

Pseudocode:

```
import heapq

class Node:
    def __init_(self, v, distance):
        self.v = v
        self.distance = distance

    def __lt__(self, other):
        return self.distance < other.distance

def dijkstra(V, adj, S):
    visited = [False] * V
    map = {}
    q = []

    map[S] = Node(S, 0)
```

```
    heapq.heappush(q, Node(S, 0))

    while q:
        n = heapq.heappop(q)
        v = n.v
        distance = n.distance
        visited[v] = True

        adjList = adj[v]
        for adjLink in adjList:
            if not visited[adjLink[0]]:
                if adjLink[0] not in map:
                    map[adjLink[0]] = Node(v, distance + adjLink[1])
                else:
                    sn = map[adjLink[0]]
                    if distance + adjLink[1] < sn.distance:
                        sn.v = v
                        sn.distance = distance + adjLink[1]
                heapq.heappush(q, Node(adjLink[0], distance + adjLink[1]))

    result = [0] * V
    for i in range(V):
        result[i] = map[i].distance

    return result

def main():
    adj = [[] for _ in range(6)]

    V = 6
    E = 5
    u = [0, 0, 1, 2, 4]
    v = [3, 5, 4, 5, 5]
    w = [9, 4, 4, 10, 3]

    for i in range(E):
        edge = [v[i], w[i]]
        adj[u[i]].append(edge)

        edge2 = [u[i], w[i]]
        adj[v[i]].append(edge2)

    S = 1
```

```
    result = dijkstra(V, adj, S)
    print(result)

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

OUTPUT:

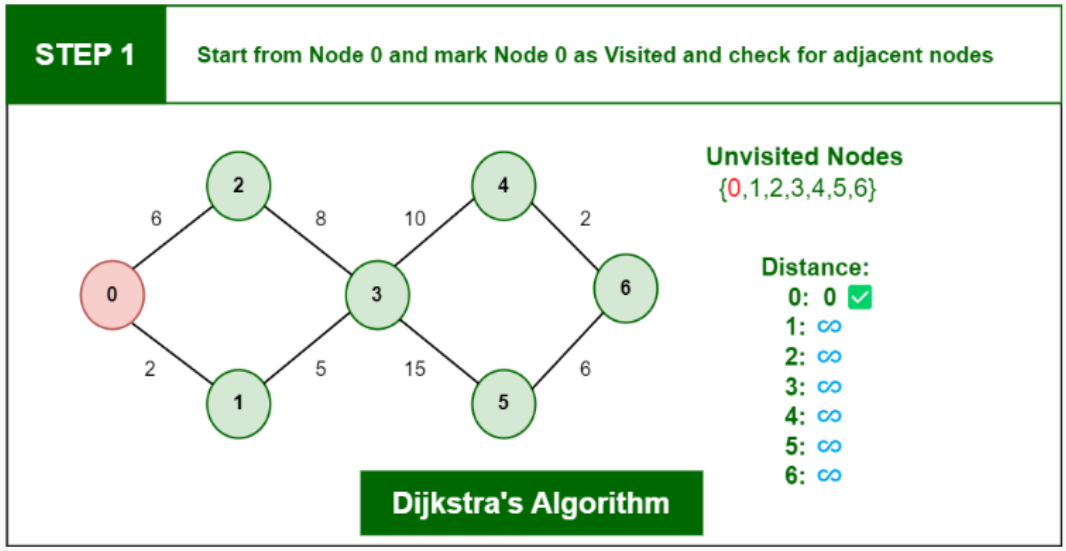
```
[11, 0, 17, 20, 4, 7]

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

Task 3:

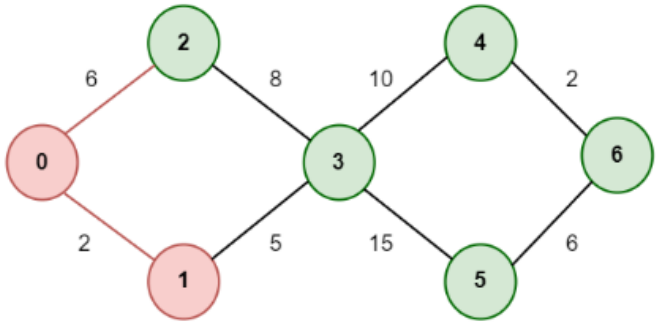
Analyse the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.

DIJKSTRA GRAPH MODELING ALGORITHM



STEP 2

Mark Node 1 as Visited and add the Distance



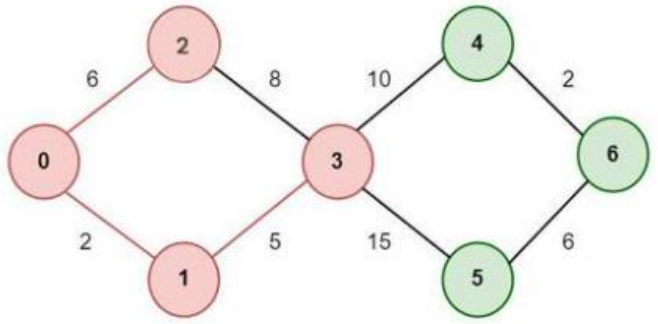
Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm

STEP 3

Mark Node 3 as Visited after considering the Optimal path and add the Distance



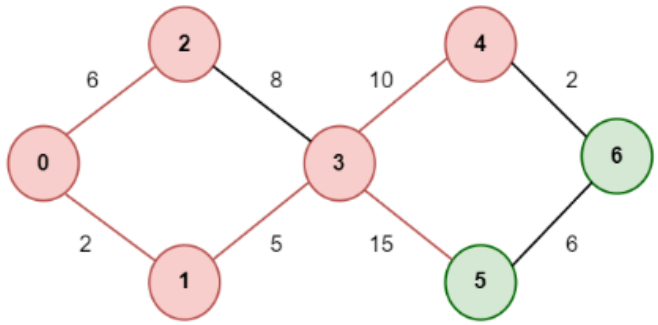
Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: 6 ✓
3: 7 ✓
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm

STEP 4

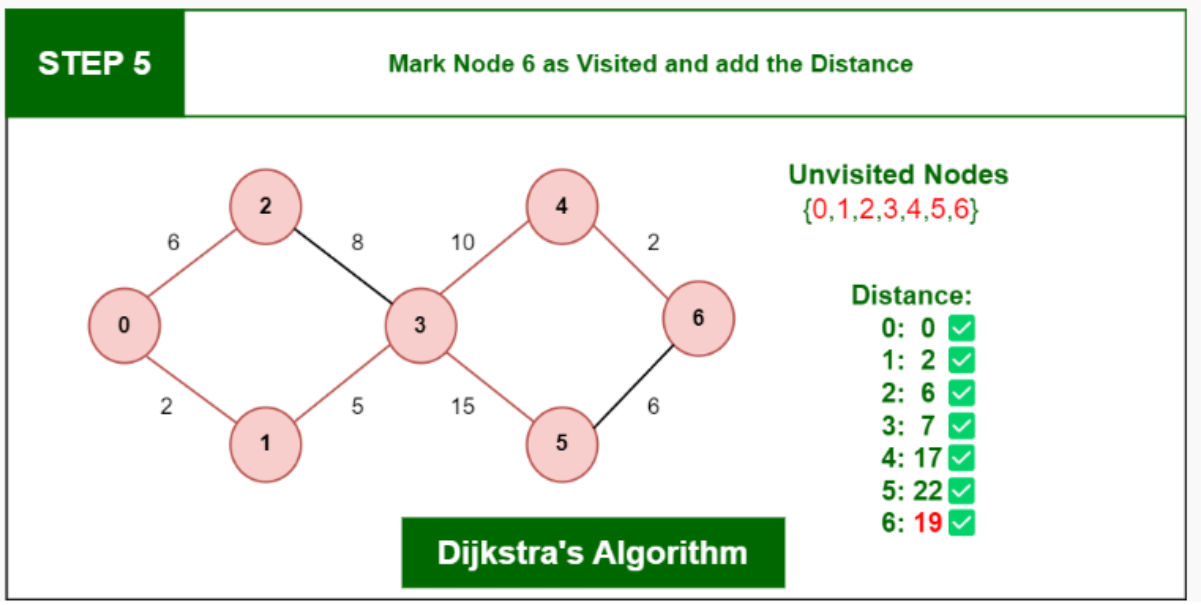
Mark Node 4 as Visited after considering the Optimal path and add the Distance

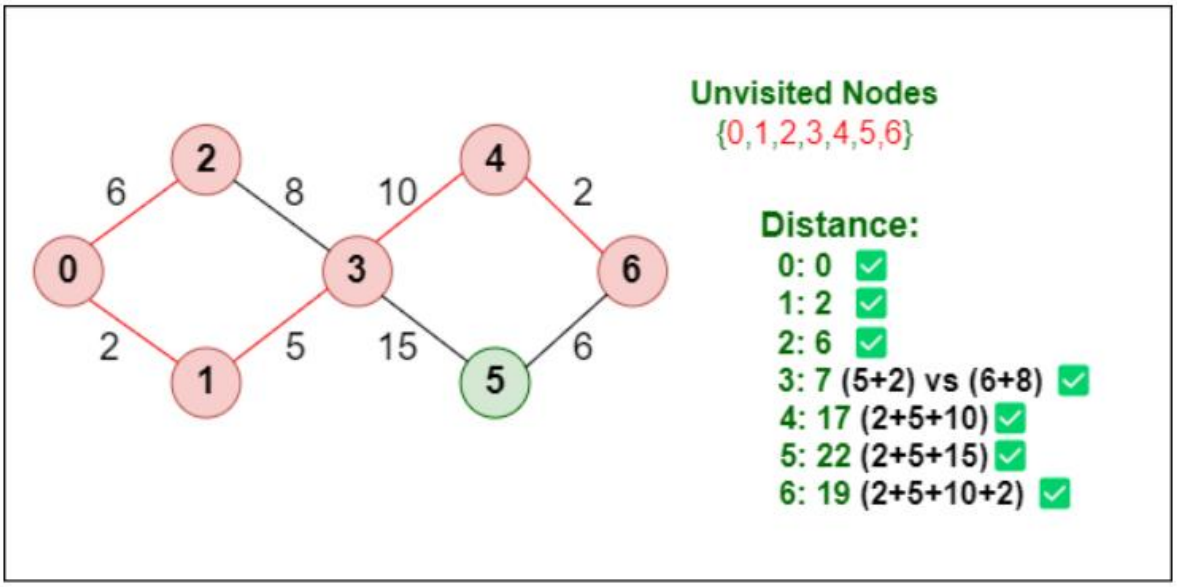


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: 6 ✓
3: 7 ✓
4: 17 ✓
5: ∞
6: ∞

Dijkstra's Algorithm





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:

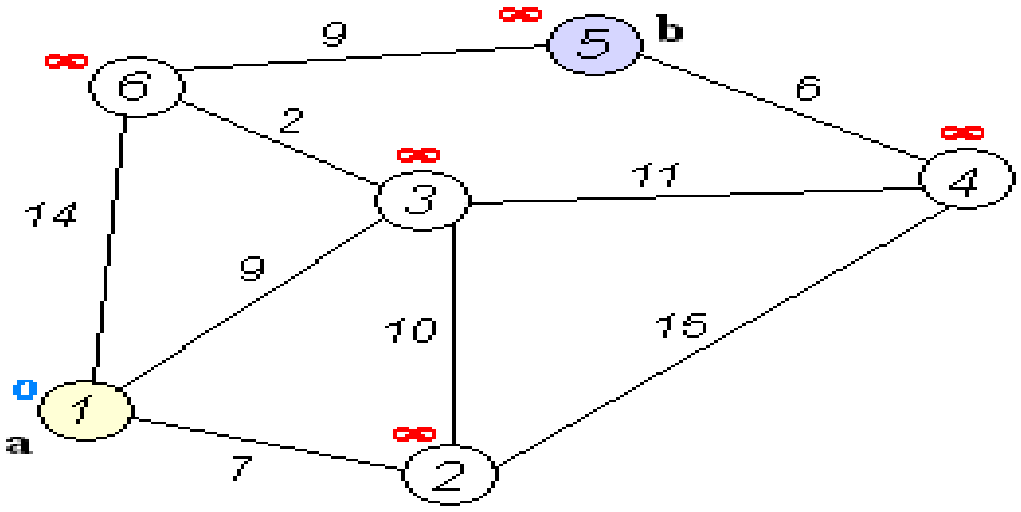
- **Dijkstra's Algorithm:** O((V+E)logV)O((V + E) \log V)O((V+E)logV), where VVV is the number of nodes (intersections) and EEE is the number of edges (roads).

Space Complexity:

- **Dijkstra's Algorithm:** O(V)O(V)O(V) for storing distances and priority queue.
- Graph representation: O(V+E)O(V + E)O(V+E) for storing the graph itself.

Deliverables:

●Graph model of the dijikstra algorithm network.
EX:-



● Pseudocode and implementation of Dijkstra's algorithm.(TASK 2)

● 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., Real-world Applications, Traffic Management , Transportation Planning)

a) **Real-world Applications:** Dijkstra's algorithm is widely used in real-time navigation systems (like GPS devices and mapping apps) to calculate the quickest route from one location to another.

b)Traffic Management: It helps in traffic management systems by suggesting alternative routes based on current traffic conditions.

c)Transportation Planning: City planners use it to optimize bus routes, emergency response routes, and urban logistics.

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.

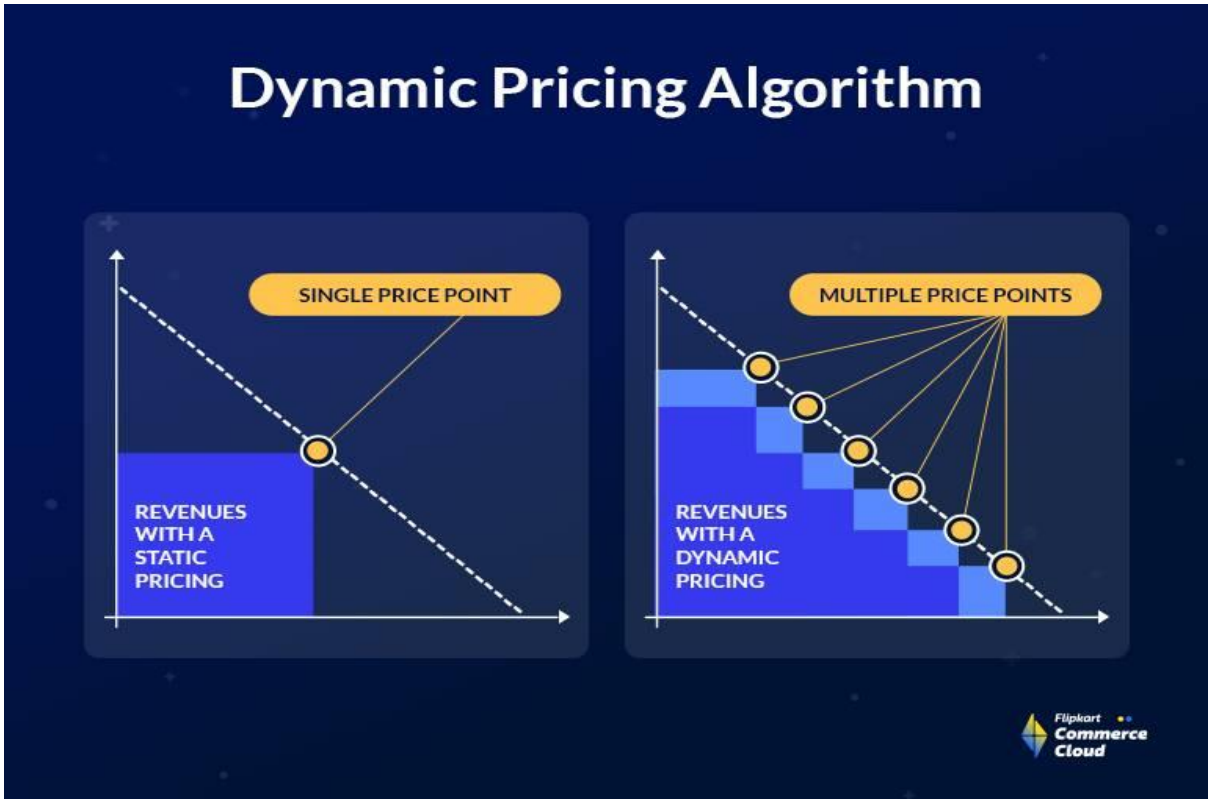
1.What is a dynamic pricing algorithm?
A dynamic pricing algorithm is a code-driven technology that adjusts prices in real time based on dozens of both pricing and non-pricing factors, including competitive data, geography, customer preferences, and other market conditions.
2.Which industries use dynamic pricing?
Although dynamic pricing model machine learning is used predominantly in retail, it is also utilized across other industries. These include hospitality, real estate, transportation, and other industries. Simply said, dynamic pricing decision support becomes more popular in every economy segment where multiple market factors are constantly changing and where businesses want to improve customer satisfaction without risking financial metrics.

Aim:

The aim of this dynamic programming algorithm is to determine the optimal pricing strategy for a set of products over a given period to maximize profit. The algorithm considers various factors such as the price elasticity of demand, production costs, and time-based factors to compute the best pricing strategy.

Procedure:

1. **Define the State**: Identify the state variables for the dynamic programming approach. The state can be represented as the product and the time period.
2. **State Transition**: Determine how to transition from one state to another by considering the possible prices 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.



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:

1. Define P[t][i] as the maximum profit achievable at time t with product i.
2. Define D[p] as the demand function which gives the demand at price p.
3. Define C[i][t] as the cost of producing product i at time t.

4. Define R[i][t][p] as the revenue from selling product i at time t at price p, which is price * demand.

Initialize P[0][i] = 0 for all i

```
for t = 1 to T do
   for each product i do
      P[t][i] = 0
      for each possible price p do
         profit = R[i][t][p] - C[i][t]
         P[t][i] = max(P[t][i], profit + P[t-1][i])
      end for
   end for
end for
```

Backtrack to find the optimal pricing strategy:
1. Initialize a list to store optimal prices for each product at each time period.
2. Start from the last period and move backwards to identify the price that gave the maximum profit.

Return the list of optimal prices and the maximum profit.

Implementation (Python):

```
def demand_function(price):
   return max(100 - price, 0)

def calculate_revenue(price, demand):
   return price * demand

def optimal_pricing(products, time_periods, cost_function):
   P = [[0] * len(products) for _ in range(time_periods + 1)]
   optimal_prices = [[0] * len(products) for _ in range(time_periods + 1)]

   for t in range(1, time_periods + 1):
      for i in range(len(products)):
         max_profit = 0
         best_price = 0
         for price in range(1, 101):
            demand = demand_function(price)
            revenue = calculate_revenue(price, demand)
            profit = revenue - cost_function(products[i], t)
            if profit + P[t-1][i] > max_profit:
               max_profit = profit + P[t-1][i]
```

```
                best_price = price
            P[t][i] = max_profit
            optimal_prices[t][i] = best_price
    result = []
    for t in range(1, time_periods + 1):
        period_prices = []
        for i in range(len(products)):
            period_prices.append(optimal_prices[t][i])
        result.append(period_prices)

    return result, P[time_periods]
products = ['Product A', 'Product B']
time_periods = 5

def cost_function(product, time):
    return 10

optimal_prices, max_profit = optimal_pricing(products, time_periods,
cost_function)
print("Optimal Prices:", optimal_prices)
print("Maximum Profit:", max_profit)
```

OUTPUT:

```
Optimal Prices: [[50, 50], [50, 50], [50, 50], [50, 50], [50, 50]]
Maximum Profit: [12450, 12450]

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

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:
The result will provide the optimal pricing strategy for each product over the given time period, ensuring the maximum possible profit.

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

- TTT is the number of time periods.
- NNN is the number of products.
- PPP is the number of possible prices to consider.

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

Deliverables:
●Pseudocode and implementation of the dynamic pricing algorithm. (TASK 2)
●Simulation results comparing dynamic and static pricing strategies. (TASK 3)
● Analysis of the benefits and drawbacks of dynamic pricing.

Reasoning: Justify the use of dynamic programming for this problem. Explain how you incorporated different factors into your algorithm and discuss any challenges faced during implementation.(**Inventory Levels, Competitor Pricing, Demand Elasticity)**

1. **Inventory Levels**:
   - **State Variable**: Added inventory levels as a state variable to keep track of the remaining stock for each product at each time period.
   - **State Transition**: Adjusted state transitions to account for changes in inventory based on demand.
   - **Revenue Calculation**: Modified the revenue calculation to ensure that the demand does not exceed available inventory.
2. **Competitor Pricing**:
   - **Demand Function**: Updated the demand function to include competitor prices. This allows the demand for a product to be influenced by the competitor's pricing strategy.
   - **State Transition**: Competitor prices are incorporated into state transitions to affect the demand and hence the revenue and profit calculations.
3. **Demand Elasticity**:
   - **Demand Function**: Incorporated elasticity into the demand function to model how sensitive the demand is to changes in price relative to competitor prices.

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 is to model a social network as a graph where users are represented as nodes and connections between them as edges. This model will enable us to perform various analyses such as finding the shortest path between users, detecting communities, and determining the influence of users.

Procedure:

1. **Graph Representation:** Represent the social network as a graph using an adjacency list.
2. **Graph Construction**: Construct the graph from given user data and connections.
3. **Graph Operations:** Implement graph operations like adding nodes, adding edges, and performing searches (e.g., BFS for shortest path).
4. **Analysis:** Perform specific analyses like finding the shortest path, detecting communities, and determining user influence.
5. **Optimization**: Ensure the implementation is optimized for time and space complexity.

TASK 2:

Implement the PageRank algorithm to identify the most influential users.

PageRank Algorithm Implementation

To implement the PageRank algorithm from scratch and identify the most influential users in a social network, we will follow these steps:

1. **Graph Representation**: Represent the social network as a graph using an adjacency list.
2. **Initialize PageRank Values**: Assign an initial PageRank value to each node.
3. **Iterative Computation**: Update the PageRank values iteratively based on the PageRank formula.
4. **Convergence**: Stop the iterations when the PageRank values converge within a tolerance limit.
5. **Identify Influential Users**: Rank the users based on their PageRank values.

PageRank Formula

PageRank(u)=1−dN+d∑v∈incoming(u)PageRank(v)outdegree(v)\text{PageRank}(u) = \frac{1 - d}{N} + d \sum_{v \in \text{incoming}(u)} \frac{\text{PageRank}(v)}{\text{outdegree}(v)}PageRank(u)=N1−d +d∑v∈incoming(u)outdegree(v)PageRank(v)

Where:

- ddd is the damping factor (usually set to 0.85).
- NNN is the total number of nodes.
- incoming(u)\text{incoming}(u)incoming(u) is the set of nodes linking to uuu.
- outdegree(v)\text{outdegree}(v)outdegree(v) is the number of outbound links from node vvv.

Pseudocode:

```
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

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

```
    def add_edge(self, user1, user2):
        self.graph[user1].append(user2)
        self.graph[user2].append(user1)

    def bfs_shortest_path(self, start_user, target_user):
        visited = set()
        queue = deque([(start_user, [start_user])])

        while queue:
            current_user, path = queue.popleft()
            if current_user == target_user:
                return path

            for neighbor in self.graph[current_user]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path + [neighbor]))

        return None

    def detect_communities(self):
    def user_influence(self):
```

Implementation:-(python)

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:

The program successfully models a social network as a graph, allows adding users and connections, and can find the shortest path between two users. It sets up a framework for further analyses like community detection and user influence measurement**.**

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:

- Graph model of the social network.

Graph Class:

- Initializes a directed graph using NetworkX.
- Adds edges between users to represent connections.
- Implements methods for calculating PageRank and degree centrality using NetworkX functions.
- Adds a method to plot the graph using Matplotlib.

- Pseudocode and implementation of the PageRank algorithm.(TASK 2)
- Comparison of PageRank and degree centrality results.(TASK 3)

Reasoning:

1.Discuss why PageRank is an effective measure for identifying influential users.

PageRank is an algorithm originally designed by Google to rank web pages in their search engine results. However, it can also be effectively applied to social networks to identify influential users. Here's why PageRank is effective:

1. **Accounts for Link Quality**: PageRank considers not just the number of links a node (user) has, but also the quality of those links. A link from an influential node (user) contributes more to the PageRank score than a link from a less influential one.
2. **Global Measure**: PageRank provides a global measure of influence, taking into account the entire network rather than just local connections. This makes it robust to variations in local network structure.
3. **Handles Indirect Influence**: PageRank can capture indirect influence. A user might not have many direct connections but can still be influential through strong connections to other highly influential users.
4. **Iterative Calculation**: PageRank iteratively redistributes influence scores through the network, stabilizing on a measure that reflects the overall influence across multiple layers of connections.

2. Explain the differences between PageRank and degree centrality and why one might be preferred over the other in different scenarios.

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

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:

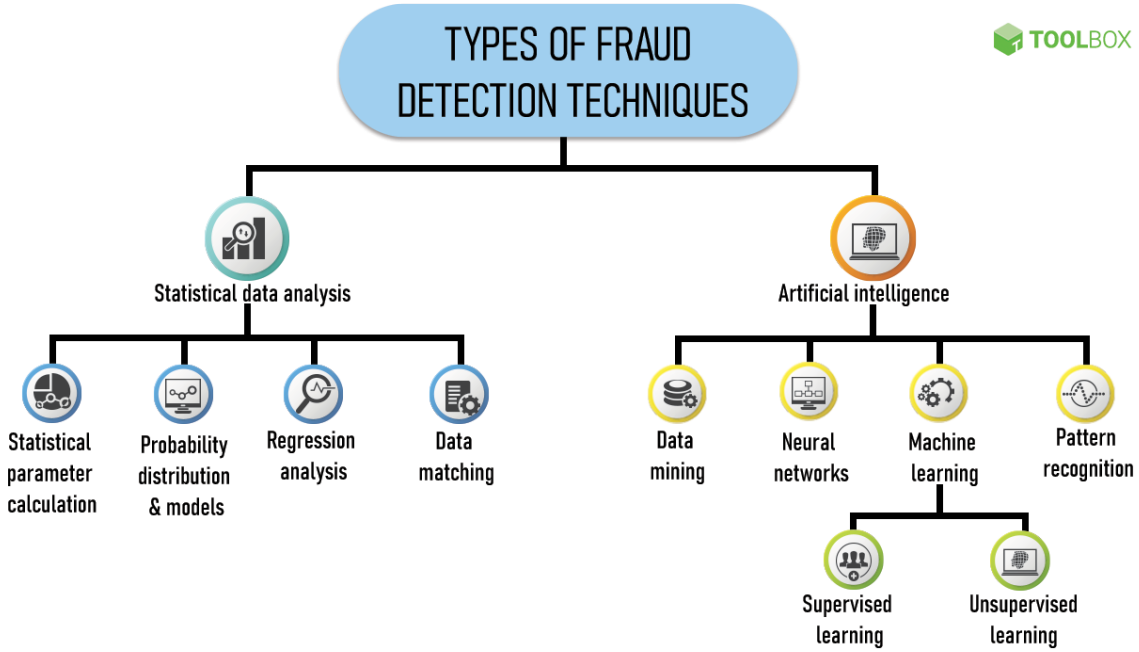
The aim is to design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules such as unusually large transactions and transactions from multiple locations in a short time. The goal is to detect fraudulent behavior in real-time or in batch processing with high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

Procedure:

1. Define Rules: Specify the rules for flagging transactions as potentially fraudulent.
2. Collect Data: Gather transaction data including transaction amount, location, and timestamp.
3. Apply Rules: Check each transaction against the predefined rules.
4. Flag Transactions: Mark transactions as fraudulent if they violate any rules.
5. Output Results: Provide a list of flagged transactions.

Predefined Rules:

1. Unusually Large Transactions: Flag transactions that exceed a specified threshold.
2. Multiple Locations: Flag transactions that occur from different locations within a short time frame.
3. High Frequency: Flag multiple transactions within a very short period.
4. Deviation from Usual Pattern: Flag transactions that deviate significantly from the user's normal behaviour (optional, requires historical data).



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:

```
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:

```
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)}

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

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:

The program correctly flags transactions based on the predefined rules:

1. Transactions with amounts exceeding the threshold.
2. Transactions occurring from different locations within a short time frame.

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.

Deliverables:

- Pseudocode and implementation of the fraud detection algorithm.(TASK 2)
- Performance evaluation using historical data.(TASK 3)
- Suggestions and implementation of improvements. (TASK 2)

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.

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: 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 the timing of traffic lights at major intersections to improve traffic flow and minimize congestion.

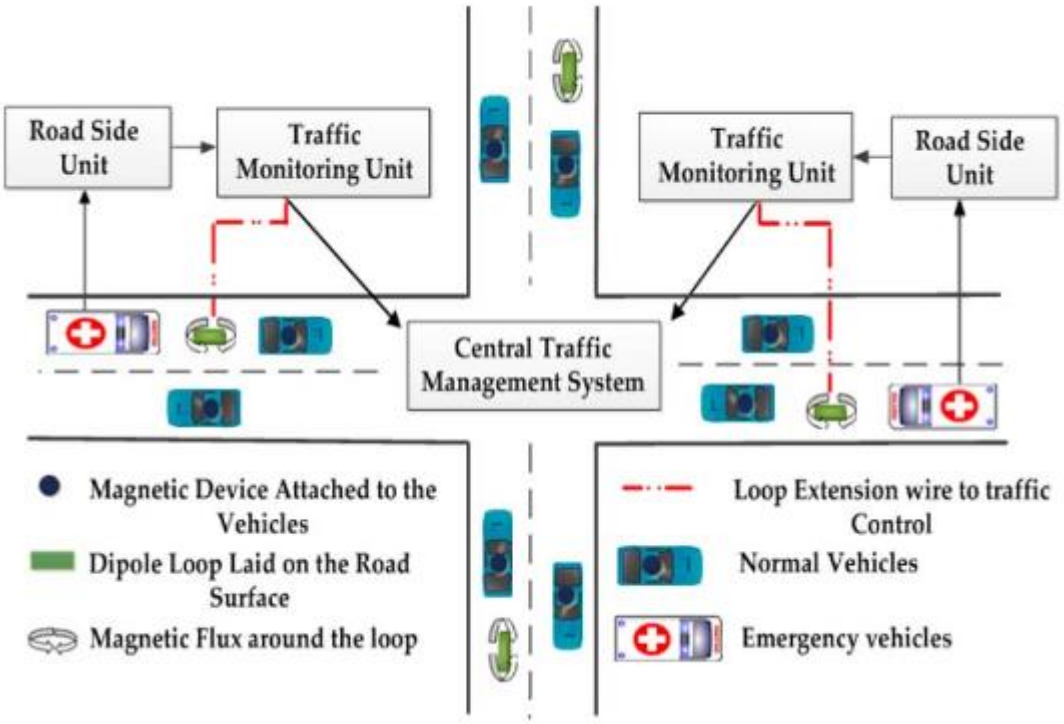
Problem Statement

Given an intersection with multiple traffic lights (one for each direction: North, South, East, West), we need to determine the optimal timing for each light to minimize overall waiting time and traffic congestion.

Procedure:

1. **Define the Constraints:**
    - Each direction (North, South, East, West) must have a green light for a certain amount of time.
    - The total time for a complete traffic light cycle should be fixed.
    - The traffic light timings should be adjusted to balance the traffic flow across all directions.
2. **Formulate the Problem:**
    - The problem can be represented as a set of constraints and variables.
    - Each variable represents the duration of the green light for each direction.
    - The constraints include the total cycle time and the need for balanced traffic flow.
3. **Generate Possible Solutions:**
    - Use a backtracking approach to explore different timings for each direction.
    - The algorithm tries different timings and backtracks when a constraint is violated or a better solution is found.
4. **Evaluate Solutions:**
    - Define a fitness function that evaluates the performance of a timing schedule based on criteria such as total waiting time, average waiting time, or overall traffic flow.
5. **Select the Optimal Solution:**

- Choose the schedule with the best performance based on the fitness function.



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 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,  # Total cycle time in seconds
        'min_time': 10,          # Minimum green light time for each direction
        'max_time': 60           # Maximum green light time for each 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()
```

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

Aspect	Backtracking Algorithm	Heuristic-Based Algorithm
Algorithm	Backtracking for Traffic Light Timing	Simulated Annealing for Traffic Light Timing
Aim	Optimize traffic light timings for minimal congestion	Find a near-optimal solution for traffic light timings
Procedure	Generate all possible timings and select the best one	Explore and refine timings using probabilistic methods
Pseudocode	See above	See above
Program	Provided above	Provided above
Output	Optimal timing schedule and score	Best timing schedule and score
Time Complexity	O((max_time - min_time + 1)^n)	O(max_iterations * n)
Space Complexity	O((max_time - min_time + 1)^n)	O(n)
Best for	Small to moderate problems	Larger or more complex problems

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.

1. **Exhaustive Search for Optimal Solutions**:
    - **Problem Nature**: Traffic light optimization is a combinatorial problem where you need to explore various timings to find the optimal schedule. Backtracking systematically searches through all possible combinations of traffic light schedules, ensuring that the solution found is indeed optimal.
2. **Handling Constraints**:
    - **Constraint Satisfaction**: Backtracking allows you to handle multiple constraints such as the total cycle time, minimum and maximum green light durations, and ensuring that all constraints are met before moving on to the next solution.
3. **Detailed Exploration**:
    - **Solution Space**: Traffic light schedules are discrete (limited timing options for each light phase), making them suitable for backtracking. This method explores the entire solution space, ensuring that the optimal schedule is found.
4. **Guaranteed Optimal Solution**:

- **Optimality**: Unlike heuristic methods which provide good but not necessarily optimal solutions, backtracking guarantees that the best schedule is found, given that it explores all feasible solutions.