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.

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 analyze 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.

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:

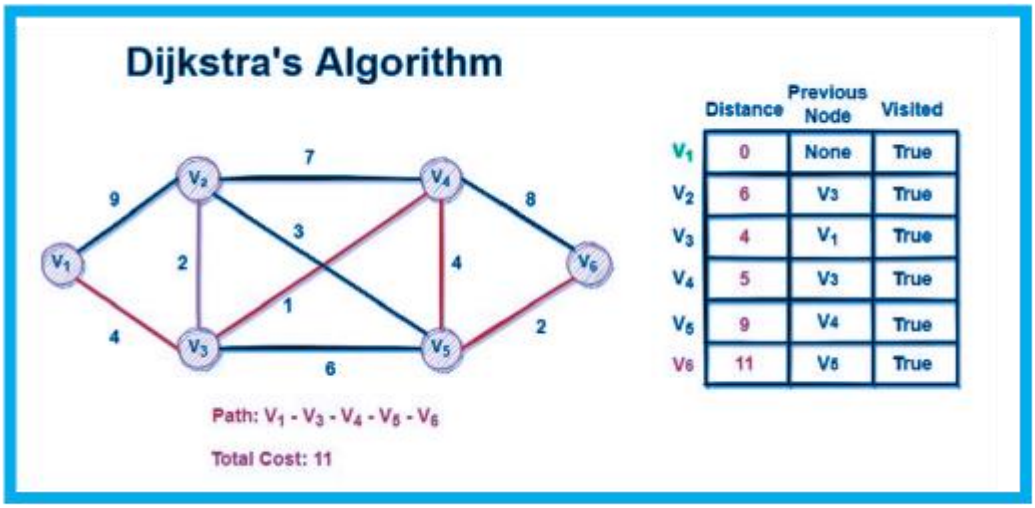
```
[11, 0, 17, 20, 4, 7]

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

Task 3:

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

DIJKSTRA ALGORITHM GRAPH MODELING:



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 city's road network.
- 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., **Efficiency, Optimality, Scalability**)

Efficiency:

- Dijkstra's algorithm efficiently finds the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights. This is ideal for modeling travel times in a city's road network where travel times (weights) are typically non-negative.

Optimality:

- The algorithm guarantees that the shortest path is found, ensuring that delivery routes from the central warehouse to various locations are optimal in terms of travel time.

Scalability:

- The algorithm can handle large graphs, which is crucial for urban road networks with many intersections (nodes) and roads (edges).

Problem 2: Dynamic Pricing Algorithm for E-commerce

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

TASK 1:

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

Aim:

The aim of this 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

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.

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.

Optimization Nature:

- The problem requires finding an optimal solution (maximizing profit) over a given period, considering various constraints and factors. DP excels in such optimization problems.

Overlapping Subproblems:

- The problem involves computing profits over multiple time periods, where the profit at each period depends on decisions made in previous periods. These overlapping subproblems make DP an appropriate technique.

Complex Dependencies:

- Inventory levels, competitor pricing, and demand elasticity create a complex dependency structure. DP handles such dependencies efficiently by storing intermediate results and reusing them.

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.

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)

```
from collections import defaultdict, deque
```

```
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):

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


    users = ['Alice', 'Bob', 'Charlie', 'David', 'Eve']
```

```
    for user in users:
        social_network.add_node(user)


    connections = [('Alice', 'Bob'), ('Alice', 'Charlie'), ('Bob', 'David'),
('Charlie', 'David'), ('David', 'Eve')]
    for user1, user2 in connections:
        social_network.add_edge(user1, user2)

    start_user = 'Alice'
    target_user = 'Eve'
    path = social_network.bfs_shortest_path(start_user, target_user)
    print(f"Shortest path from {start_user} to {target_user}: {path}")
```

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.

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.

Resistance to Manipulation:

- Because PageRank considers the quality and not just the quantity of connections, it is more resistant to manipulation. Users cannot easily inflate their influence by simply increasing the number of connections; they need to be connected to other influential users.

Versatility:

- PageRank can be applied to various types of networks beyond web pages, such as social networks, citation networks, and even biological networks. This versatility underscores its robustness as a measure of influence.

Relevance in Real-World Networks:

- Real-world networks often exhibit power-law distributions and clustering, characteristics well-handled by PageRank. Influential users (or nodes) in such networks typically have a high PageRank because they connect clusters or serve as hubs in the network.

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 behavior (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.

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.

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.

Immediate Decisions: Fraud detection requires making decisions based on current transactions or activities. Greedy algorithms can quickly process transactions and flag suspicious activity in real-time.

Fast Execution: Greedy algorithms generally have low time complexity, which is crucial for real-time systems where decisions must be made within milliseconds to prevent fraud.

Example: For a credit card transaction, a greedy approach might involve checking a few key features (such as transaction amount and location) to determine if further investigation is needed, rather than running a complex model that takes more time.

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.

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.

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.

Incremental Adjustments: Backtracking works incrementally, which aligns well with real-time scenarios where decisions need to be adjusted based on ongoing traffic conditions.

Adaptation to Changes: Traffic conditions change frequently (accidents, roadworks), and backtracking can adapt to these changes by re-evaluating previous decisions and exploring alternative options.

Example: Adjusting traffic light timings based on real-time traffic flow data is a scenario where backtracking can evaluate different timing options to find the best schedule under changing conditions.