Assignment: Design and Analysis of Algorithm

Due Date : July 1 2024

Problem 1: Optimizing Delivery Routes

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: To create a structured model of the city's road network using graph theory.

Procedure:

1)**Modeling the Road Network as a Graph**:

- Each intersection is represented as a node.
- Each road between intersections is represented as an edge with a weight (travel time).

2)**Using Dijkstra's Algorithm**:

- Dijkstra's algorithm is suitable here because it efficiently finds the shortest path from a source node to all other nodes in a graph with non-negative weights.

Pseudo Code:

```
function Dijkstra(Graph, source):
    dist[source] := 0
    create priority queue Q
    Q.add(source, dist[source])

    while Q is not empty:
        u := Q.extractMin()
        for each neighbor v of u:
            alt := dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] := alt
                Q.addOrUpdate(v, alt)


    return dist
```

Program:

```
import heapq

def dijkstra(graph, source):
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    priority_queue = [(0, source)]

    while priority_queue:
```

```python
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > dist[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < dist[neighbor]:
                dist[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return dist
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

source = 'A'
distances = dijkstra(graph, source)
print("Shortest distances from", source)
for node, distance in distances.items():
    print(f"To node {node}: {distance}")
```

Analysis:

Analysis:

- Time complexity: O((V+E) log V), where V is the number of nodes and E is the number of edges (roads). This due to the use of a priority queue.

- Space complexity: O(V), for storing the distance and the priority queue

Time Complexity: O((V+E)logV)
Space Complexity: O(V)
Output:

```
Shortest distances from A
To node A: 0
To node B: 1
To node C: 3
To node D: 4
Press any key to continue . . .
```

Result: The Program is successfully executed without errors

Task 2:

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

Procedure:

1)**Graph Representation**:

- Represent the road network as a graph where nodes represent intersections or locations, and edges represent roads with weights representing travel times.

2)**Priority Queue**:

- Use a priority queue (min-heap) to efficiently fetch the node with the smallest known distance during the algorithm's execution.

3)**Initialization**:

- Initialize distances from the source (central warehouse) to all other nodes as infinity, except for the source node itself which is set to 0.

4)**Algorithm Execution**:

- Repeat until all nodes have been processed:
  - Extract the node with the smallest distance from the priority queue.
  - Update distances to its neighbors if a shorter path is found through the current node.
  - Push updated distances and neighbors back into the priority queue.

Pseudo Code:

```
function Dijkstra(Graph, source):
    dist[source] := 0
    create priority queue Q
    Q.add(source, dist[source])

    while Q is not empty:
        u := Q.extractMin()
        for each neighbor v of u:
            alt := dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] := alt
                Q.addOrUpdate(v, alt)

    return dist
```

Program:

```python
import heapq

def dijkstra(graph, source):
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

def main():
    graph = {
        'Warehouse': {'A': 5, 'B': 7},
        'A': {'C': 2, 'D': 4},
        'B': {'D': 1},
        'C': {'D': 3},
        'D': {}
    }
    source = 'Warehouse'
    shortest_distances = dijkstra(graph, source)

    print(f"Shortest paths from '{source}':")
    for node, distance in shortest_distances.items():
        print(f"To '{node}': {distance} units")

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

Analysis:

• 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 nodes (intersections or locations) and E is the number of edges (roads)

• Each node is extracted from the priority queue once ($O(V\log V)$), and for each node, each each is relaxed at most once ($O(E)$).

• Space complexity:

• The space complexity is $O(V+E)$ to storing of graph and the priority queue

Time Complexity: O((V+E)logV)

Space Complexity: O(V+E)

Output:

```
Shortest paths from 'Warehouse':
To 'Warehouse': 0 units
To 'A': 5 units
To 'B': 7 units
To 'C': 7 units
To 'D': 8 units
Press any key to continue . . .
```

Result: The code is executed successfully without any errors

Task 3: Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.

Aim: to determine the shortest paths from a central warehouse to multiple delivery locations in a road network represented as a graph

Procedure:

1)**Graph Representation**:

- Represent the road network as a weighted graph where nodes represent intersections or locations, and edges represent roads with weights (travel times).

2)**Dijkstra's Algorithm**:

- Use Dijkstra's algorithm due to its efficiency in finding the shortest path from a single source node to all other nodes in graphs with non-negative weights.

3)**Implementation**:

- Implement Dijkstra's algorithm using a priority queue (min-heap) for efficient extraction of the node with the smallest known distance.
- Update distances to neighboring nodes as shorter paths are discovered.
- Continue until all nodes have been processed or no shorter paths can be found.

Pseudo Code:

```
function Dijkstra(Graph, source):
    dist[source] := 0
    create priority queue Q
    Q.add(source, dist[source])

    while Q is not empty:
        u := Q.extractMin()
        for each neighbor v of u:
            alt := dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] := alt
                Q.addOrUpdate(v, alt)

    return dist
```

Program:

```python
import heapq

def dijkstra(graph, source):
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]
```

```python
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

def main():
    graph = {
        'Warehouse': {'A': 5, 'B': 7},
        'A': {'C': 2, 'D': 4},
        'B': {'D': 1},
        'C': {'D': 3},
        'D': {}
    }

    source = 'Warehouse'
    shortest_distances = dijkstra(graph, source)

    print(f"Shortest paths from '{source}':")
    for node, distance in shortest_distances.items():
        print(f"To '{node}': {distance} units")

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

Analysis:

3) **Analysis of Efficiency:**

Alternate Algorithms:

* Bellman-Ford algorithm: usefull when there are negative edge weights
* Bidirectional Dijkstra's Algorithm: can be faster in some scenarios
* A Search Algorithm: uses heuristics to guid the search towards destination

Improvements:

- **Graph preprocessing:** If graph is static, preprocessing techniques like precomputing shortest paths using Floyd-warshall algorithm can be beneficial
- **Heap optimization:** using Fibonacci heaps or other advanced priority queue structures can further optimize the performance of Dijkstra's algorithm.

Time complexity: O((V+E)logV))

Space Complexity: O(V+E)

Output:

```
Shortest paths from 'Warehouse':
To 'Warehouse': 0 units
To 'A': 5 units
To 'B': 7 units
To 'C': 7 units
To 'D': 8 units
Press any key to continue . . .
```

Result: The code is executed successfully without any errors

Problem 2: Dynamic Pricing Algorithm for E-commerce

Task 1: Design a dynamic programming algorithm to determine the optimal pricing strategy for a set of products over a given period

Aim: To design a dynamic programming algorithm to maximize total revenue or profit by strategically setting optimal prices for a set of products over a given period.

Procedure:

1.define state variables:

- DP[t][i] represents the maximum profit up to time t considering the pricing of product i

2.Base case:

- DP[0][i] = 0 for all products I .

3.Reccurence Relation:

- For each product I at time t, calculate the potential profit by choosing different prices and update the DP table accordingly.
- Consider demand elasticity and constraints in the calculation of profit.

Pseudo code:

```
def optimal_pricing_strategy (prices, demand, costs, T, N):
      DP = [[0 for _ in range(N)] for _ in range(T+1)]
      for t in range (1, T+1):
           for i in range(N):
                max_profit = 0
                for p in prices[i]
                    d = demand[i](p, t)
                    profit = (p - costs[i]) * d
                    max_profit = max(max_profit, profit + DP[t-1][i])
                   DP[t][i] = max_profit
                  optimal_profit = max (DP[T])
    return optimal_profit
```

program:

```
def optimal_pricing_strategy (prices, demand_funcs, costs, T, N):
    DP = [[0 for _ in range(N)] for _ in range(T+1)]
    for t in range (1, T+1):
        for i in range(N):
            max_profit = 0
            for p in prices[i]:
                d = demand_funcs[i](p, t)
                profit = (p - costs[i]) * d
                max_profit = max (max_profit, profit + DP[t-1][i])
            DP[t][i] = max_profit
        optimal_profit = max(DP[T])
    return optimal_profit
```

```python
prices = [[10, 15, 20], [5, 10, 15]]
demand_funcs = [
    lambda p, t: 100 - 2*p + t,
    lambda p, t: 200 - 3*p + 2*t ]
costs = [5, 3]
T = 10
N = 2
optimal_profit = optimal_pricing_strategy(prices, demand_funcs, costs, T, N)
print (f"Optimal Profit: {optimal_profit}")
```

output:

```
Optimal Profit: 19920
Press any key to continue . . .
```

Analysis:

Analysis:

- Time complexity:
  - The time complexity is dominated by the nested loops iterating over time steps ('T'), products ('N'), and prices ('k' where 'k' is the average number of prices per product). Therefore, the time complexity is approximately O(T*N*k)

- space complexity:
  - The space complexity is O(T*N), primarily due to the 'DP' array storing maximum profits for each product up to each time step.

Time complexity: O(T*N*K)

Space complexity: O(T*N)

Result: The code executed successfully without errors

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

Aim: The aim of this algorithm is to optimize the pricing strategy for our products by dynamically adjusting prices based on real time inventory levels, competitor pricing and demand elasticity.

Procedure:

1.Define state variables:

- DP[t][i][s] represent the maximum profit up to time t considering the pricing of product I with s units of inventory remaining

2.Base case:

- DP[0][i][s]= 0 for all products I and inventory levels s.

3.**Reccurence Relation**:

- For each product I at time t and inventory level s, calculate the potential profit by choosing different prices and update the DP table accordingly:

  DP[t][i][s]= max(profit at price p +DP[t-1][i][s-demand]

Pseudo code:

```
def optimal_pricing_strategy(prices, demand, costs, T, N, inventory,
competitor_prices):
  DP = [[[0 for _ in range(inventory[i]+1)] for _ in range(N)] for _ in range(T+1)]
       for t in range(1, T+1):
        for i in range(N):
            for s in range(inventory[i]+1):
                max_profit = 0
                for p in prices[i]:
                    d = demand[i](p, t, competitor_prices[i])
                    if d <= s:
                        profit = (p - costs[i]) * d
                        max_profit = max (max_profit, profit + DP[t-1][i][s-d])
                DP[t][i][s] = max_profit
        optimal_profit = max (max (DP[T][i]) for i in range(N))
    return optimal_profit
```

Program:

```
def optimal_pricing_strategy(prices, demand_funcs, costs, T, N, inventory,
competitor_prices):
    DP = [[[0 for _ in range(max(inventory)+1)] for _ in range(N)] for _ in
range(T+1)]
    for t in range(1, T+1):
```

```python
        for i in range(N):
            for s in range(inventory[i]+1):
                max_profit = 0
                for p in prices[i]:
                    d = demand_funcs[i](p, t, competitor_prices[i])
                    if d <= s:  # Ensure demand does not exceed current inventory
                        profit = (p - costs[i]) * d
                        max_profit = max (max_profit, profit + DP[t-1][i][s-d])
                DP[t][i][s] = max_profit
    optimal_profit = max (max (DP[T][i]) for i in range(N))
    return optimal_profit
prices = [[10, 15, 20], [5, 10, 15]]
demand_funcs = [
    lambda p, t, cp: max (0, 100 - 2*p + t - 0.5*cp),
    lambda p, t, cp: max (0, 200 - 3*p + 2*t - 0.3*cp)]
costs = [5, 3]
T = 10
N = 2
inventory = [50, 100]
competitor_prices = [12, 8]
optimal_profit = optimal_pricing_strategy (prices, demand_funcs, costs, T, N,
inventory, competitor_prices)
print (f"Optimal Profit: {optimal_profit}")
```

Output:

```
Optimal Profit: 0
Press any key to continue . . .
```

Analysis:

2.2

Analysis:

- Time complexity:

  - The time complexity is approxiamately O(T * N * I * K), where:

    - 'T' is the time horizon
    - 'N' is the number of products
    - 'I' is the maximum inventory capacity among all products
    - 'K' is the average number of prices per product

- space complexity:

  - The space complexity is O(T*N*I), primarily due to the 'DP' array storing maximum profits for each product, time step, and inventory state

Time complexity: O(T*N*I*K)
Space complexity: O(T*N*I)

Result: The code executed successfully without any errors

<u>Task 3:</u> Test your algorithm with simulated data and compare its performance with a simple static pricing strategy

<u>Aim:</u> To maximize revenue or profit by leveraging real-time market conditions while comparing its performance against a simple static pricing strategy

<u>Procedure:</u>

1.intialization and setup:

- Define products and assign initial prices to each product

2.simulation:

- Simulate sales using dynamic prices and compare results with static pricing strategy.

3.Evaluation:

- Analyse performance metrics to determine the effectiveness of dynamic pricing

4.adjustment:

- Fine-tune the algorithm based on evaluation findings to optimize pricing strategy

<u>Pseudo code</u>:

```
demand_trends):
current_prices = initial_prices
 while market_conditions: function dynamic_pricing_algorithm (products,
initial_prices, competitor_prices,
        update_demand_trends(demand_trends)
        update_competitor_prices(competitor_prices)

        for product in products:
            new_price = calculate_new_price (product, current_prices, demand_trends,
competitor_prices)
            new_price = apply_price_constraints(new_price)
            current_prices[product] = new_price
    return current_prices
function compare_performance (static_prices, dynamic_prices):
    revenue_static = simulate_sales(static_prices)
    revenue_dynamic = simulate_sales(dynamic_prices)
    performance_comparison = analyze_performance (revenue_static, revenue_dynamic)
    return performance_comparison
```

<u>Program:</u>

```
import random
def update_demand_trends(products):
    for product in products:
        products[product]['demand'] += random.uniform(-5, 5)
```

```python
def update_competitor_prices(products):
    for product in products:
        products[product]['competitor_price'] += random.uniform (-2, 2)
def calculate_new_price (current_price, demand, competitor_price):
    new_price = current_price * (1 + 0.1 * (competitor_price - current_price)) * (1 + 0.05 * demand)
    return new_price
def simulate_sales (prices, demand_trends):
    total_revenue = 0
    for product, price in prices.items ():
        demand = demand_trends[product]['demand']
        sales_volume = demand * random.uniform (0.8, 1.2)
        revenue = sales_volume * price
        total_revenue += revenue
    return total_revenue
def main ():
    products = {
        'product1': {'price': 50, 'demand': 100, 'competitor_price': 45},
        'product2': {'price': 30, 'demand': 150, 'competitor_price': 28}
    }
    static_prices = {product: products[product]['price'] for product in products}
    dynamic_prices = {}
    for product, info in products.items():
        current_price = info['price']
        demand = info['demand']
        competitor_price = info['competitor_price']
        new_price = calculate_new_price (current_price, demand, competitor_price)
        dynamic_prices[product] = new_price

    revenue_static = simulate_sales (static_prices, products)
    revenue_dynamic = simulate_sales (dynamic_prices, products)

    print (f"Static Pricing Revenue: ${revenue_static}")
    print (f"Dynamic Pricing Revenue: ${revenue_dynamic}")

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

output:

```
Static Pricing Revenue: $8732.457530787684
Dynamic Pricing Revenue: $50043.89521940074
Press any key to continue . . .
```

Analysis:

2.3

Analysis

Time complexity :

update - demand - trends (products) : O(n)

update - competitor - prices (products) : O(n)

calculate - new - price : O(1)

simulate - sales (prices) : O(n)

main() : O(n)

overall Time complexity : O(n)

space complexity :

update - demand - trends (products) : O(1)

update - competitor - price (products) : O(1)

calculate - new - price : O(1)

simulate - sales (prices) : O(1)

main() : O(n)

overall space complexity : O(n)

Time complexity: **O(n)**

Space complexity: **O(n)**

Result: The code executed successfully

Program 3: Social Network Analysis (Case Study)

Task 1: Model the social network as a graph where users are nodes and connections are edges.

Aim: To analyze the structural properties and dynamics of a social network by modelling it as a graph, identifying key nodes, communities, and understanding how information propagates within the network.

Procedure:

1)Initialize the Graph:

Create a new graph object. Use nx.Graph() for an undirected graph or nx.DiGraph() for a directed graph.

2)Collect Data:

Prepare a list of users (nodes).

Prepare a list of connections (edges) between users.

3)Create Nodes:

Add the users as nodes to the graph.

4)Create Edges:

Add the connections as edges to the graph.

5)Visualize the Graph:

Set up the visualization using matplotlib.

Draw the graph with labels and customize the appearance (e.g., node color, edge color, node size)

Pseudo Code:

```
class SocialNetwork:
    initialize():
        users = {}
    add_user(user):
        if user not in users:
            users[user] = set()
    remove_user(user):
        if user in users:
            for friend in users[user]:
                users[friend].remove(user)
            del users[user]
    add_connection(user1, user2):
        if user1 in users and user2 in users:
            users[user1].add(user2)
            users[user2].add(user1)
    remove_connection(user1, user2):
```

```
        if user1 in users and user2 in users:
            users[user1].remove(user2)
            users[user2].remove(user1)
    get_friends(user):
        if user in users:
            return users[user]
    are_connected(user1, user2):
        if user1 in users and user2 in users:
            return user2 in users[user1]
    user_exists(user):
        return user in users
```

Program:

```
class SocialNetwork:
    def __init__(self):
        self.users = {}
    def add_user(self, user):
        if user not in self.users:
            self.users[user] = set()
    def remove_user(self, user):
        if user in self.users:
            for friend in self.users[user]:
                self.users[friend].discard(user)
            del self.users[user]
    def add_connection(self, user1, user2):
        if user1 in self.users and user2 in self.users:
            self.users[user1].add(user2)
            self.users[user2].add(user1)
    def remove_connection(self, user1, user2):
        if user1 in self.users and user2 in self.users:
            self.users[user1].discard(user2)
            self.users[user2].discard(user1)
    def get_friends(self, user):
        return self.users.get(user, set())
    def are_connected(self, user1, user2):
        return user1 in self.users and user2 in self.users and user2 in
self.users[user1]
    def user_exists(self, user):
        return user in self.users

if __name__ == "__main__":
    network = SocialNetwork()
    network.add_user("sunny")
    network.add_user("Bob")
    network.add_user("Charlie")
    network.add_connection("sunny", "Bob")
    network.add_connection("sunny", "Charlie")
    print(f"sunny friends: {network.get_friends('sunny')}")
    print(f"Are sunny and Bob connected? {network.are_connected('sunny', 'Bob')}")
    print(f"Are Bob and Charlie connected? {network.are_connected('Bob',
'Charlie')}")
    network.remove_connection("sunny", "Bob")
    print(f"Are sunny and Bob connected after removal?
{network.are_connected('sunny', 'Bob')}")
    network.remove_user("Charlie")
    print(f"Does Charlie exist in the network? {network.user_exists('Charlie')}")
    print(f"sunny friends after Charlie removal: {network.get_friends('sunny')}")
```

Analysis:

3.1

<u>Analysis</u>

Time complexity:

* Adding Node : O(1)
* Adding Edge : O(1) to O(log N)
* Finding Neighbours : O(1) to O(N)
* Traversal (BFS/DFS) : O(N+E)

Space complexity:

* Nodes and edges : O(N+E)

Output:

```
sunny friends: {'Charlie', 'Bob'}
Are sunny and Bob connected? True
Are Bob and Charlie connected? False
Are sunny and Bob connected after removal? False
Does Charlie exist in the network? False
sunny friends after Charlie removal: set()
Press any key to continue . . .
```

Time complexity: O(M+N)

Space complexity: O(U+F)

Result: The code executed successfully without errors

Task 2: Implement the PageRank algorithm to identify the most influential users.

Aim: To implement the PageRank algorithm to identify the most influential users in a social network modeled as a graph, where users are represented as nodes and connections are represented as edges.

Procedure:

1. Initialize: Assign each node an initial PageRank value.
2. Iterate: Update the PageRank value of each node based on the PageRank values of its incoming connections.
3. Convergence: Repeat the iteration until the PageRank values converge (i.e., the change in values is less than a small threshold).

Pseudo Code:

```
1. Initialize:
   a. N = number of nodes in the graph
   b. pagerank = {node: 1/N for each node in the graph}
   c. new_pagerank = copy of pagerank
2. Iterate for max_iterations:
   a. For each node in graph:
       i. Set rank_sum = 0
       ii. For each incoming node in graph:
           - If node is in graph[incoming]:
               - Add pagerank[incoming] / len(graph[incoming]) to rank_sum
       iii. Update new_pagerank[node] = (1 - d)/N + d * rank_sum
   b. Calculate diff = sum(abs(new_pagerank[node] - pagerank[node]) for each node in
pagerank)
   c. If diff < tol:
       - Break the loop
   d. Copy new_pagerank to pagerank
3. Return pagerank
```

Program:

```
import numpy as np
def pagerank(graph, d=0.85, max_iterations=100, tol=1.0e-6):
    N = len(graph)
    pagerank = {node: 1/N for node in graph}
    new_pagerank = pagerank.copy()
    for iteration in range(max_iterations):
        for node in graph:
            rank_sum = 0
            for incoming in graph:
                if node in graph[incoming]:
                    rank_sum += pagerank[incoming] / len(graph[incoming])
            new_pagerank[node] = (1 - d)/N + d * rank_sum
        diff = sum(abs(new_pagerank[node] - pagerank[node]) for node in pagerank)
        if diff < tol:
            break
        pagerank = new_pagerank.copy()
    return pagerank
if __name__ == "__main__":
    graph = {
        'A': ['B', 'C'],
        'B': ['C'],
```

```
        'C': ['A'],
        'D': ['C']
    }
    pagerank_values = pagerank(graph)
    print("PageRank values:", pagerank_values)
```

Analysis:

Time complexity:

• Initialization: O(N), N is the number of nodes

• iteration update: O(N+E), E is the number of edges

• Convergence: The number of iterations can vary but is typically logarithmic in nature concerning the number of nodes and edges due to convergence properties of the algorithm

Space complexity:

• Graph representation: O(N+E), N is the space for storing nodes
E is the space for storing nodes

• Rank page scores: O(N), as Each node requires storage for its PageRank score

Output:

```
PageRank values: {'A': 0.37252644684091407, 'B': 0.19582422337929592, 'C': 0
    .39414932977979, 'D': 0.037500000000000006}

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

Time complexity: O(max_iterations*|V^2|)
Space Complexity: O(|V|+|E|)
Result: The code executed successfully without errors

Task 3: Compare the results of PageRank with a simple degree centrality measure.

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

Procedure:

1. PageRank Algorithm :-

- Initialize each node's PageRank score.
- Iteratively update PageRank scores based on neighbor contributions until convergence.
- Output the final PageRank scores.

2. Degree Centrality:

- Calculate the number of connections (degree) for each node.
- Normalize the degree by dividing by N-1 (where N is the total number of nodes) to get the degree centrality score.

3. Comparison:

- Compare the rankings of users based on PageRank scores and degree centrality scores.
- Evaluate correlation or differences in identifying influential users.

Pseudo Code:

```
Procedure PageRank(Graph G):
    Initialize PageRank scores for all nodes
    while not converged:
        for each node v in G:
            newPageRank[v] = (1 - d) + d * sum(PageRank[u] / outDegree[u] for u ->
v)
        if PageRank scores converge:
            break
        else:
            Update PageRank scores
Procedure DegreeCentrality(Graph G):
    for each node v in G:
        degreeCentrality[v] = degree(v) / (N - 1)  // N is total number of nodes
Procedure CompareResults(PageRankScores, DegreeCentralityScores):
    Compare rankings or correlation between PageRankScores and
DegreeCentralityScores
```

Program:

```
import networkx as nx
G = nx.DiGraph()
G.add_edges_from([(1, 2), (1, 3), (2, 3), (3, 1)])
pagerank_scores = nx.pagerank(G, alpha=0.85)
degree_centrality_scores = nx.degree_centrality(G)
pagerank_sorted = sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True)
```

```
degree_centrality_sorted = sorted(degree_centrality_scores.items(), key=lambda x:
x[1], reverse=True)
print("PageRank Scores:")
for node, score in pagerank_sorted:
    print(f"Node {node}: {score}")
print("\nDegree Centrality Scores:")
for node, score in degree_centrality_sorted:
    print(f"Node {node}: {score}")
```

Output:

```
PageRank Scores:
Node 3: 0.3873015873015873
Node 1: 0.33730158730158727
Node 2: 0.2753968253968254

Degree Centrality Scores:
Node 1: 1.5
Node 3: 1.0
Node 2: 0.5
```

Analysis:

Time complexity:

- For each node, the algorithm counts the number of edges.
- Counting the degree of all nodes takes O(E) time because each edge is considered once
- Thus, the total time complexity is O(E)

Space complexity:

→ The algorithm needs to store the graph and the degree of each node

→ storing the graph takes O(N+E) space

→ storing the degree of each node. takes O(N) space

→ Thus, the total space complexity is O(N+E)

Time complexity: O(E)

Space Complexity:O(N+E)

Result: The code executed successfully without errors

Program 4: Fraud Detection in Financial Transactions

Task 1: Design a greedy algorithm to flag potentially fraudulent transactions based on asset of predefined rules

Aim: To, detect potentially fraudulent transactions using a set of predefined rules to flag transactions that exhibit unusual patterns, such as being unusually large or originating from multiple locations within a short time frame

Procedure:

1.**Define Rules**: Establish the criteria for flagging transactions as potentially fraudulent.

2.Data Input: Gather transaction data including:

- Transaction ID
- Amount
- Timestamp
- Location (e.g., IP address or geolocation)
- User ID

3.Initialization: Create data structures to keep track of user transaction patterns and recent transactions.

4.Iterate Through Transactions: For each transaction, apply the predefined rules to check if it should be flagged as potentially fraudulent.

- If the transaction amount exceeds the threshold, flag it.
- If there are multiple transactions from different locations for the same user within a short period, flag it.
- If the transaction time is unusual, flag it.

5.Flag Transactions: Store the flagged transactions in a list or database.

Pseudo Code:

```
Define RULE_AMOUNT_THRESHOLD as a large transaction threshold
Define RULE_LOCATION_TIME_THRESHOLD as a short time period threshold
Initialize flagged_transactions as an empty list
Initialize user_transactions as an empty dictionary
FOR each transaction IN transactions:
    Extract user_id, amount, timestamp, and location from the transaction
    IF amount > RULE_AMOUNT_THRESHOLD:
          Append {transaction_id, reason: "Large amount"} to flagged_transactions
     IF user_id is not in user_transactions:
          Initialize user_transactions[user_id] as an empty list
Append (timestamp, location) to user_transactions[user_id]
```

```
 Filter user_transactions[user_id] to only include transactions within
RULE_LOCATION_TIME_THRESHOLD of the current transaction timestamp
  Extract unique locations from the filtered transactions
   IF the number of unique locations > 1:
          Append {transaction_id, reason: "Multiple locations"} to
flagged_transactions
     IF transaction occurs at an unusual time (e.g., late night):
           Append {transaction_id, reason: "Unusual time"} to flagged_transactions
RETURN flagged_transactions
```

Program:

```
from datetime import datetime, timedelta
RULE_AMOUNT_THRESHOLD = 1000.0
RULE_LOCATION_TIME_THRESHOLD = timedelta(minutes=30)
def flag_fraudulent_transactions(transactions):
    flagged_transactions = []
    user_transactions = {}
    for txn in transactions:
        user_id = txn['user_id']
        amount = txn['amount']
        timestamp = txn['timestamp']
        location = txn['location']
        transaction_id = txn['transaction_id']
        if amount > RULE_AMOUNT_THRESHOLD:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Large amount"   })
        if user_id not in user_transactions:
            user_transactions[user_id] = []
        user_transactions[user_id].append((timestamp, location))
        recent_transactions = [
            t for t in user_transactions[user_id]
            if t[0] > timestamp - RULE_LOCATION_TIME_THRESHOLD ]
        unique_locations = set(t[1] for t in recent_transactions)
        if len(unique_locations) > 1:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Multiple locations"   })
        if timestamp.hour < 6 or timestamp.hour > 22:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Unusual time"     })
    return flagged_transactions
transactions = [
    {"transaction_id": "T1", "amount": 5000.0, "timestamp": datetime(2024, 6, 29,
10, 30), "location": "New York", "user_id": "U1"},
    {"transaction_id": "T2", "amount": 300.0, "timestamp": datetime(2024, 6, 29, 10,
45), "location": "Los Angeles", "user_id": "U1"},
    {"transaction_id": "T3", "amount": 50.0, "timestamp": datetime(2024, 6, 29, 23,
0), "location": "New York", "user_id": "U2"},]

flagged_transactions = flag_fraudulent_transactions(transactions)
for ft in flagged_transactions:
    print(ft)
```

Analysis:

4.1

Analysis

Time Complexity:

- Initializing structure O(1)
- Iterating through transaction O(n)

∴ The time complexity is O(n)

Space complexity:

- O(n) + O(n) = O(n)

Output:

```
{'transaction_id': 'T1', 'reason': 'Large amount'}
{'transaction_id': 'T2', 'reason': 'Multiple locations'}
{'transaction_id': 'T3', 'reason': 'Unusual time'}
Press any key to continue . . .
```

Time complexity:O(n)

Space complexity:O(n)

Result: The code is executed without any errors

Task 2: Evaluate the algorithm's performance using historical transaction data and calculate metrics such as precision, recall, and F1 score.

Aim: To evaluate the performance of the algorithm designed to flag potentially fraudulent transactions by using historical transaction data. The performance will be measured using metrics such as precision, recall, and F1 score.

Procedure:

1. **Prepare Historical Transaction Data**:Obtain a dataset with transactions, including labels indicating whether each transaction is fraudulent or not.

2.**Apply the Algorithm**:Use the designed greedy algorithm to flag transactions in the historical data.

3. **Compare with Ground Truth:**Compare the flagged transactions with the actual labels to calculate the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).

4.**Calculate Metrics**:

- **Precision**: Precision=TPTP+FP\text{Precision} = \frac{TP}{TP + FP}Precision=TP+FPTP
- **Recall**: Recall=TPTP+FN\text{Recall} = \frac{TP}{TP + FN}Recall=TP+FNTP
- **F1 Score**: F1 Score=2×Precision×RecallPrecision+Recall\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}F1 Score=2×Precision+RecallPrecision×Recall

Pseudo Code:

```
1. Define RULE_AMOUNT_THRESHOLD as a large transaction threshold
2. Define RULE_LOCATION_TIME_THRESHOLD as a short time period threshold
3. Define UNUSUAL_HOUR_START and UNUSUAL_HOUR_END as the range of unusual
transaction hours
4. Initialize flagged_transactions as an empty list
5. Initialize user_transactions as an empty dictionary
6. FOR each transaction IN transactions:
    7. Extract user_id, amount, timestamp, location, and transaction_id from the
transaction
    8. IF amount > RULE_AMOUNT_THRESHOLD:
        9. Append {transaction_id, reason: "Large amount"} to flagged_transactions
    10. IF user_id is not in user_transactions:
        11. Initialize user_transactions[user_id] as an empty list
    12. Append (timestamp, location) to user_transactions[user_id]
    13. Filter user_transactions[user_id] to only include transactions within
RULE_LOCATION_TIME_THRESHOLD of the current transaction timestamp
    14. Extract unique locations from the filtered transactions
    15. IF the number of unique locations > 1:
```

```
        16. Append {transaction_id, reason: "Multiple locations"} to
flagged_transactions
    17. IF timestamp.hour < UNUSUAL_HOUR_START OR timestamp.hour > UNUSUAL_HOUR_END:
        18. Append {transaction_id, reason: "Unusual time"} to flagged_transactions
19. Initialize TP, FP, TN, and FN as 0
20. FOR each transaction IN transactions:
    21. IF transaction is flagged AND is fraudulent:
        22. Increment TP
    23. ELSE IF transaction is flagged AND is not fraudulent:
        24. Increment FP
    25. ELSE IF transaction is not flagged AND is not fraudulent:
        26. Increment TN
    27. ELSE IF transaction is not flagged AND is fraudulent:
        28. Increment FN

29. Calculate Precision = TP / (TP + FP)
30. Calculate Recall = TP / (TP + FN)
31. Calculate F1 Score = 2 * (Precision * Recall) / (Precision + Recall)
32. RETURN Precision, Recall, F1 Score
```

Program:

```
from datetime import datetime, timedelta
from collections import defaultdict
RULE_AMOUNT_THRESHOLD = 1000.0
RULE_LOCATION_TIME_THRESHOLD = timedelta(minutes=30)
UNUSUAL_HOUR_START = 22
UNUSUAL_HOUR_END = 6
def flag_fraudulent_transactions(transactions):
    flagged_transactions = []
    user_transactions = defaultdict(list)
    for txn in transactions:
        user_id = txn['user_id']
        amount = txn['amount']
        timestamp = txn['timestamp']
        location = txn['location']
        transaction_id = txn['transaction_id']
        if amount > RULE_AMOUNT_THRESHOLD:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Large amount"
            })
        user_transactions[user_id].append((timestamp, location))
        recent_transactions = [
            t for t in user_transactions[user_id]
            if t[0] > timestamp - RULE_LOCATION_TIME_THRESHOLD
        ]
        unique_locations = set(t[1] for t in recent_transactions)
        if len(unique_locations) > 1:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Multiple locations"
            })
        if timestamp.hour >= UNUSUAL_HOUR_START or timestamp.hour <
UNUSUAL_HOUR_END:
            flagged_transactions.append({
                "transaction_id": transaction_id,
```

```
                "reason": "Unusual time"
            })
    return flagged_transactions
def evaluate_algorithm(transactions, flagged_transactions):
    TP = FP = TN = FN = 0
    flagged_transaction_ids = set(txn["transaction_id"] for txn in
flagged_transactions)
    for txn in transactions:
        transaction_id = txn['transaction_id']
        is_fraudulent = txn['is_fraudulent']
        if transaction_id in flagged_transaction_ids and is_fraudulent:
            TP += 1
        elif transaction_id in flagged_transaction_ids and not is_fraudulent:
            FP += 1
        elif transaction_id not in flagged_transaction_ids and not is_fraudulent:
            TN += 1
        elif transaction_id not in flagged_transaction_ids and is_fraudulent:
            FN += 1
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision +
recall) > 0 else 0
    return precision, recall, f1_score
transactions = [
    {"transaction_id": "T1", "amount": 5000.0, "timestamp": datetime(2024, 6, 29,
10, 30), "location": "New York", "user_id": "U1", "is_fraudulent": True},
    {"transaction_id": "T2", "amount": 300.0, "timestamp": datetime(2024, 6, 29, 10,
45), "location": "Los Angeles", "user_id": "U1", "is_fraudulent": False},
    {"transaction_id": "T3", "amount": 50.0, "timestamp": datetime(2024, 6, 29, 23,
0), "location": "New York", "user_id": "U2", "is_fraudulent": True},
]
flagged_transactions = flag_fraudulent_transactions(transactions)
precision, recall, f1_score = evaluate_algorithm(transactions, flagged_transactions)
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1_score}")
```

Analysis:

Analysis:-

- Time complexity
  - Flagging transactions : O(N), N is the number of transactions
  - Evaluating algorithm · O(N), same as above
  - overall time complexity : O(N)

- space complexity!
  - Additional space for storing flagged transaction and user transactions
  - space complexity is primarily O(N) due to storing transaction details and flagged transactions.

Output:

```
Precision: 0.6666666666666666
Recall: 1.0
F1 Score: 0.8
Press any key to continue . . .
```

Time complexity: O(n)

Space complexity: O(n)

Result: The code executed successfully without any errors

Task 3: Suggest and implement potential improvements to the algorithm

Aim: To improve the algorithm for flagging potentially fraudulent transactions

Procedure:

1.Reduce Redundant Checks: Instead of repeatedly filtering transactions for each user, maintain a sliding window of recent transactions. Use efficient data structures like a deque to maintain the recent transactions within the given time threshold.

2.Utilize Efficient Data Structures: Use sets for locations to automatically handle uniqueness and improve lookup times. Use dictionaries to store user-specific information, which allows for O(1) average-time complexity for insertions and lookups.

3.Parallel Processing: If the dataset is large, consider parallel processing to divide the workload and process multiple transactions simultaneously.

4.Improve Rule Checking Logic: Precompute certain values, such as unusual hours, to avoid redundant calculations.

Pseudo Code:

```
flag_fraudulent_transactions(transactions):
    flagged_transactions = []
    user_transactions = {}
    for txn in transactions:
        user_id = txn.user_id
        amount = txn.amount
        timestamp = txn.timestamp
        location = txn.location
        transaction_id = txn.transaction_id
        if amount > RULE_AMOUNT_THRESHOLD:
            flagged_transactions.append({transaction_id, "Large amount"})
        if user_id not in user_transactions:
            user_transactions[user_id] = deque()
        while user_transactions[user_id] and user_transactions[user_id][0][0] <
timestamp - RULE_LOCATION_TIME_THRESHOLD:
            user_transactions[user_id].popleft()
        user_transactions[user_id].append((timestamp, location))
        unique_locations = set(loc for _, loc in user_transactions[user_id])
        if len(unique_locations) > 1:
            flagged_transactions.append({transaction_id, "Multiple locations"})
        if timestamp.hour >= UNUSUAL_HOUR_START or timestamp.hour <
UNUSUAL_HOUR_END:
            flagged_transactions.append({transaction_id, "Unusual time"})
    return flagged_transaction
evaluate_algorithm(transactions, flagged_transactions):
    TP = 0
    FP = 0
    TN = 0
    FN = 0
```

```
    flagged_transaction_ids = set(txn.transaction_id for txn in
flagged_transactions)
    for txn in transactions:
        transaction_id = txn.transaction_id
        is_fraudulent = txn.is_fraudulent
        if transaction_id in flagged_transaction_ids and is_fraudulent:
            TP += 1
        elif transaction_id in flagged_transaction_ids and not is_fraudulent:
            FP += 1
        elif transaction_id not in flagged_transaction_ids and not is_fraudulent:
            TN += 1
        elif transaction_id not in flagged_transaction_ids and is_fraudulent:
           FN += 1
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision +
recall) > 0 else 0
    return precision, recall, f1_score
```

Program:

```
from datetime import datetime, timedelta
from collections import defaultdict, deque
RULE_AMOUNT_THRESHOLD = 1000.0
RULE_LOCATION_TIME_THRESHOLD = timedelta(minutes=30)
UNUSUAL_HOUR_START = 22
UNUSUAL_HOUR_END = 6
def flag_fraudulent_transactions(transactions):
    flagged_transactions = []
    user_transactions = defaultdict(deque)
    for txn in transactions:
        user_id = txn['user_id']
        amount = txn['amount']
        timestamp = txn['timestamp']
        location = txn['location']
        transaction_id = txn['transaction_id']
        if amount > RULE_AMOUNT_THRESHOLD:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Large amount"
            })
        while user_transactions[user_id] and user_transactions[user_id][0][0] <
timestamp - RULE_LOCATION_TIME_THRESHOLD:
            user_transactions[user_id].popleft()
        user_transactions[user_id].append((timestamp, location))
        unique_locations = set(loc for _, loc in user_transactions[user_id])
        if len(unique_locations) > 1:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Multiple locations"
            })
        if timestamp.hour >= UNUSUAL_HOUR_START or timestamp.hour <
UNUSUAL_HOUR_END:
            flagged_transactions.append({
                "transaction_id": transaction_id,
                "reason": "Unusual time"
            })
```

```
    return flagged_transactions
def evaluate_algorithm(transactions, flagged_transactions):
    TP = FP = TN = FN = 0
    flagged_transaction_ids = set(txn["transaction_id"] for txn in
flagged_transactions)
    for txn in transactions:
        transaction_id = txn['transaction_id']
        is_fraudulent = txn['is_fraudulent']
        if transaction_id in flagged_transaction_ids and is_fraudulent:
            TP += 1
        elif transaction_id in flagged_transaction_ids and not is_fraudulent:
            FP += 1
        elif transaction_id not in flagged_transaction_ids and not is_fraudulent:
            TN += 1
        elif transaction_id not in flagged_transaction_ids and is_fraudulent:
            FN += 1

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision +
recall) > 0 else 0
    return precision, recall, f1_score
transactions = [
    {"transaction_id": "T1", "amount": 5000.0, "timestamp": datetime(2024, 6, 29,
10, 30), "location": "New York", "user_id": "U1", "is_fraudulent": True},
    {"transaction_id": "T2", "amount": 300.0, "timestamp": datetime(2024, 6, 29, 10,
45), "location": "Los Angeles", "user_id": "U1", "is_fraudulent": False},
    {"transaction_id": "T3", "amount": 50.0, "timestamp": datetime(2024, 6, 29, 23,
0), "location": "New York", "user_id": "U2", "is_fraudulent": True},
]
flagged_transactions = flag_fraudulent_transactions(transactions)
precision, recall, f1_score = evaluate_algorithm(transactions, flagged_transactions)
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1_score}")
```

Analysis:

4.3

Analysis:

- Time complexity:
  - Flagging transactions : O(N)
  - Evaluating algorithm : O(N)
  - overall Time complexity : O(N)

- Space complexity:
  - Additional space for storing flagged transactions and user transactions
  - space complexity is primarily O(N) due to storing transaction details and flagged transactions

Output:

```
Precision: 0.6666666666666666
Recall: 1.0
F1 Score: 0.8
Press any key to continue . . .
```

Time complexity: O(n)

Space Complexity: O(n)

Result: The code executed successfully without any errors

Program 5: Real-Time Traffic Management System

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 using a backtracking algorithm to minimize overall traffic congestion and maximize traffic flow efficiency.

Procedure:

1)**Define the Problem**: Identify intersections, traffic flows, and constraints such as road capacity and traffic patterns.

2)**Formulate States**: Each state represents a potential configuration of traffic light timings at intersections.

3)**Define Constraints**: Consider factors like maximum green light durations, traffic load balancing, and safety requirements.

4)**Backtracking Approach**:

- Start with an initial configuration of traffic light timings.
- Explore neighboring configurations (adjacent states) to improve traffic flow.
- Evaluate each configuration based on predefined metrics (e.g., minimize average waiting time, maximize throughput).
- Use pruning techniques to avoid exploring configurations that violate constraints or do not improve upon current solutions.
- Continue until all configurations are explored or a satisfactory solution is found.

5)**Optimization Criteria**: Measure the effectiveness of each configuration using simulation or analytical models that calculate traffic flow metrics.

6)**Implementation**: Implement the backtracking algorithm using appropriate data structures and algorithms to efficiently explore and evaluate configurations.

Pseudo Code:

```
function optimizeTrafficLights(intersections):
    best_configuration = None
    current_configuration = initial_configuration

    function backtrack(current_configuration):
        if isComplete(current_configuration):
            evaluate(current_configuration)
            if meetsConstraints(current_configuration):
```

```
                updateBest(current_configuration)
            return

        for next_configuration in generateNeighbors(current_configuration):
            backtrack(next_configuration)

    backtrack(current_configuration)
    return best_configuration
```

Program:

```
def optimize_traffic_lights(intersections, initial_configuration):
    best_configuration = None
    current_configuration = initial_configuration

    def is_complete(configuration):
        return all(intersection in configuration for intersection in intersections)

    def evaluate(configuration):
        pass

    def meets_constraints(configuration):
        return True

    def is_better(new_configuration, current_best):
        return True

    def generate_neighbors(configuration):

        pass

    def backtrack(current_configuration):
        nonlocal best_configuration
        if is_complete(current_configuration):
            evaluate(current_configuration)
            if meets_constraints(current_configuration):
                if best_configuration is None or is_better(current_configuration,
best_configuration):
                    best_configuration = current_configuration.copy()
            return

        for next_configuration in generate_neighbors(current_configuration):
            backtrack(next_configuration)

    backtrack(current_configuration)
    return best_configuration

intersections = ['Intersection1', 'Intersection2', 'Intersection3']
initial_configuration = {'Intersection1': 'Green', 'Intersection2': 'Red',
'Intersection3': 'Yellow'}
optimized_configuration = optimize_traffic_lights(intersections,
initial_configuration)
print("Optimized Traffic Light Configuration:", optimized_configuration)
```

Analysis:

5.1

Analysis:

- Time complexity:

The time complexity depend on the size of the state space explored by the backtracking algorithm. In worst case scenarios, it can be exponential $O(b^d)$ where b is the branching factor.

- space complexity:

The space complexity primarily depends on the depth of recursion, auxiliary data structures used and potentially the size of the state space. it is typically $O(d)$ in the worst case.

Output:

```
Optimized Traffic Light Configuration: {'Intersection1': 'Green', 'Intersection2': 'Red', 'Intersection3': 'Yellow'}
Press any key to continue . . .
```

Time complexity: $O(b^d)$

Space complexity: $O(d)$

Result: The code executed successfully

Task 2: Simulate the algorithm on a model of the city's traffic network and measure its impact on traffic flow.

Aim: To implement and simulate the backtracking algorithm for optimizing traffic light timings on a model of the city's traffic network, and measure its impact on traffic flow metrics such as average travel time, throughput, and congestion levels.

Procedure:

1)**Traffic Network Model**: Construct a simplified model of the city's traffic network using graph theory, where intersections are nodes and roads are edges with weights representing travel times or congestion levels.

2)**Initial Traffic Light Configuration**: Define an initial configuration of traffic light timings for intersections.

3)**Backtracking Algorithm Integration**: Integrate the backtracking algorithm (designed earlier) to optimize traffic light timings based on the traffic network model.

4)**Simulation Setup**:

- Simulate the movement of vehicles through the traffic network using the optimized traffic light timings.
- Collect metrics such as average travel time, vehicle throughput (vehicles passing through intersections per unit time), and congestion levels (waiting times at intersections).

5)**Evaluation**:

- Compare the simulation results before and after applying the optimized traffic light timings.
- Analyze the impact on traffic flow metrics to determine the effectiveness of the algorithm in reducing congestion and improving traffic efficiency.

Pseudo Code:

```
function simulateAlgorithmOnTrafficNetwork():
    initializeTrafficNetwork()
    initialTrafficLightConfiguration = generateInitialConfiguration()
    optimizedTrafficLightConfiguration = optimize_traffic_lights(intersections,
initialTrafficLightConfiguration)
    applyTrafficLightConfiguration(optimizedTrafficLightConfiguration)

    runSimulation()

    evaluateTrafficFlowMetrics()
    printMetrics()

function runSimulation():
```

```
    pass

function evaluateTrafficFlowMetrics():
    pass
```

Program:

```python
import itertools

def evaluate_timing(timing):
    total_wait_time = sum(timing.values())
    max_wait_time = max(timing.values())
    return -max_wait_time

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

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

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

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

    return best_timing, best_score

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

    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()
```

Analysis:

<u>Analysis</u>:-

• <u>Time</u> <u>complexity</u>: The primary factor is exponential in terms of the range of possible times and the number of directions. However, it's manageable within reasonable bounds due to practical limits on 'min_time', 'max_time', and the number of directions typically involved in traffic light control

$$\therefore T(n) = O((max\_time - min\_time + 1)^{len(directions)}) \times len(directions)$$

• <u>Space</u> <u>complexity</u>: The space complexity is relatively low, mainly linear with respect to the number of directions.

Output:

```
Best Timing Schedule: {'North': 30, 'South': 30, 'East': 30, 'West': 30}
Best Score (negative of max wait time): -30
Press any key to continue . . .
```

Time complexity: $O((max\_time-min\_time+1^{len(directions)})*len(directions))$

Space complexity: O(len(directions))

Result: The code executed successfully without any errors

Task 3: Compare the performance of your algorithm with a fixed-time traffic light system

Aim: To compare the performance in terms of traffic flow metrics between a backtracking algorithm for optimizing traffic lights and a fixed-time traffic light system.

Procedure:

1)**Backtracking Algorithm**: Implement the backtracking algorithm to dynamically optimize traffic light timings based on traffic conditions.
2)**Fixed-Time Traffic Light System**: Define a fixed-time traffic light system where each intersection has pre-defined timings that do not change.
3)**Traffic Simulation**: Simulate traffic flow using both systems under various scenarios (e.g., peak hours, low traffic periods, emergencies).

4)**Measure Traffic Metrics**: Measure and compare traffic flow metrics such as average travel time, throughput, and congestion levels between the two systems.

Pseudo Code:

```
function optimize_traffic_lights(intersections, initial_configuration):
    best_configuration = None
    current_configuration = initial_configuration

    function is_complete(current_configuration):
        // Check if all intersections have a valid traffic light setting
        return True/False

    function evaluate(current_configuration):
        // Evaluate traffic flow metrics based on the current configuration
        return score

    function meets_constraints(current_configuration):
        // Check if the current configuration meets predefined constraints
        return True/False

    function is_better(current_configuration, best_configuration):
        // Compare configurations based on evaluation function (lower score is
better)
        return True/False

    function generate_neighbors(current_configuration):
        // Generate neighboring configurations (toggle traffic lights)
        return list_of_neighbors

    function backtrack(current_configuration):
        if is_complete(current_configuration):
            evaluate(current_configuration)
            if meets_constraints(current_configuration):
                if best_configuration is None or is_better(current_configuration,
best_configuration):
                    best_configuration = current_configuration.copy()
```

```
            return

        for next_configuration in generate_neighbors(current_configuration):
            backtrack(next_configuration)

    backtrack(current_configuration)
    return best_configuration
```

Program:

```
from collections import defaultdict, deque

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

    def add_intersection(self, intersection):
        if intersection not in self.graph:
            self.graph[intersection] = []

    def add_road(self, intersection1, intersection2):
        self.graph[intersection1].append(intersection2)
        self.graph[intersection2].append(intersection1)

    def bfs_shortest_path(self, start_intersection, target_intersection):
        visited = set()
        queue = deque([(start_intersection, [start_intersection])])

        while queue:
            current_intersection, path = queue.popleft()
            if current_intersection == target_intersection:
                return path

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

        return None

class FixedTimeTrafficLightSystem:
    def __init__(self, intersections, green_light_times):
        self.intersections = intersections
        self.green_light_times = green_light_times

    def get_green_light_time(self, intersection):
        return self.green_light_times[intersection]

# Example comparison of Dynamic vs Fixed-Time Traffic Light Systems

if __name__ == "__main__":
    # Dynamic Traffic Light System
    traffic_system = DynamicTrafficLightSystem()
```

```
    intersections = ['A', 'B', 'C', 'D', 'E']
    for intersection in intersections:
        traffic_system.add_intersection(intersection)

    roads = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('D', 'E')]
    for intersection1, intersection2 in roads:
        traffic_system.add_road(intersection1, intersection2)

    start_intersection = 'A'
    target_intersection = 'E'
    path = traffic_system.bfs_shortest_path(start_intersection, target_intersection)
    print(f"Dynamic Traffic Light System: Shortest path from {start_intersection} to
{target_intersection}: {path}")

    # Fixed-Time Traffic Light System
    intersections = ['A', 'B', 'C', 'D', 'E']
    green_light_times = {
        'A': 20,
        'B': 30,
        'C': 25,
        'D': 35,
        'E': 15
    }

    traffic_light_system = FixedTimeTrafficLightSystem(intersections,
green_light_times)

    for intersection in intersections:
        green_light_time = traffic_light_system.get_green_light_time(intersection)
        print(f"Fixed-Time Traffic Light System: Intersection {intersection}: Green
light time - {green_light_time} seconds")
```

Analysis:

Dynamic Traffic Light system

- Time complexity is O(V+E), where V is the number of vertices and E is the number of edges

- Fixed Time Traffic Light system

  - Time complexity is O(N) + O(1) where N is the number of intersections.

- Space complexity:

Dynamic Traffic Light system

- overall space complexity is O(V+E), V is the number of vertrices, E is the number of roads

Fixed Time Traffic Light System

- space complexity is O(N), where N is the number of intersections

Output:

```
Dynamic Traffic Light System: Shortest path from A to E: ['A', 'B', 'D', 'E']
Fixed-Time Traffic Light System: Intersection A: Green light time - 20 seconds
Fixed-Time Traffic Light System: Intersection B: Green light time - 30 seconds
Fixed-Time Traffic Light System: Intersection C: Green light time - 25 seconds
Fixed-Time Traffic Light System: Intersection D: Green light time - 35 seconds
Fixed-Time Traffic Light System: Intersection E: Green light time - 15 seconds
Press any key to continue . . .
```

Time complexity: O(V+E)

Space Complexity: O(n)

Result: The code is executed successfully