Assignment: Design and Analysis of Algorithms

Due Date: July 1 2024

Date: July 1 2024

Program 1: Optimizing Delivery Routes (Case study)

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. This allows for efficient route planning, optimization of traffic flow, and informed decision-making in urban planning. The goal is to improve transportation efficiency, reduce congestion, and enhance overall urban mobility and safety.

Procedure:

1. **Graph Representation:**

- Define the city's road network as a dictionary of dictionaries (road_network).

2. **Initialization:**

- Initialize a priority queue (min-heap) to keep track of nodes to explore, starting with the source node (start).

3. **Start Node:**

- Start from the specified source node (start) and initialize its distance as 0 in shortest_paths.

4. **Priority Queue Handling:**

- Repeat until all nodes have been processed or the destination node (goal) is reached

5. **Path Reconstruction:**

- Once the destination node (goal) is reached or all nodes have been processed, reconstruct the shortest path from goal back to start using the shortest_paths dictionary.

Analysis :

Time complexity :-

* Initialization $O(1)$
* while loop :-

-> visited :- $O(1)$

-> Iterating over neighbours :- $O(E)$ per node, where E is the number of edges.

-> updating shortest_path :- $O(1)$ per neighbour. $O(V)$ per iteration, where V is the number of vertices.

Time complexity $= O(V^2 + VE) \approx O(V^2)$

Assuming $E \approx V^2$.

Space complexity :-

-> Graph Representation :- $O(V+E)$

-> Shortest paths Dictionary :- $O(V)$

-> Visited set $O(V)$

$= O(V+E)$

Pseudocode:

function dijkstra(graph, start, goal)

pq <- priority queue containing (0, start)

shortest_paths <- dictionary with key start and value (None, 0)

visited <- empty set

while pq is not empty

```
    current_distance, current_node <- pq.pop()

    if current_node in visited
        continue

    visited.add(current_node)

    if current_node == goal
        break

    for next_node, weight in graph[current_node]
        if next_node in visited
            continue

        new_weight <- current_distance + weight

        if new_weight < shortest_paths.get(next_node, (None, infinity))[1]
            shortest_paths[2next_node] <- (current_node, new_weight)
            pq.push((new_weight, next_node))

if goal not in shortest_paths
    return "Route Not Possible"

path <- empty list
current_node <- goal
```

Program :

```
import heapq



road_network = {
    'A': {'B': 5, 'C': 7},
    'B': {'A': 5, 'C': 3, 'D': 4},
    'C': {'A': 7, 'B': 3, 'D': 6},
    'D': {'B': 4, 'C': 6}
}

def dijkstra(graph, start, goal):
    shortest_paths = {start: (None, 0)}
    current_node = start
    visited = set()

    while current_node != goal:
        visited.add(current_node)
        destinations = graph[current_node].items()

        for next_node, weight in destinations:
            if next_node in visited:
                continue
            new_weight = shortest_paths[current_node][1] + weight

            if shortest_paths.get(next_node, (None, float('inf')))[1] > new_weight:
```

```
            shortest_paths[next_node] = (current_node, new_weight)

        next_destinations = {node: shortest_paths[node] for node in
shortest_paths if node not in visited}
        if not next_destinations:
            return "Route Not Possible"

        current_node = min(next_destinations, key=lambda k:
next_destinations[k][1])

    path = []
    while current_node is not None:
        path.append(current_node)
        next_node = shortest_paths[current_node][0]
        current_node = next_node
    path = path[::-1]
    return path

start = 'A'
goal = 'D'
shortest_path = dijkstra(road_network, start, goal)

if shortest_path == "Route Not Possible":
    print("No route found!")
else:
    print(f"Shortest path from {start} to {goal}: {shortest_path}")
```

total_weight = sum(road_network[shortest_path[i]][shortest_path[i + 1]] for i in range(len(shortest_path) - 1))

print(f"Total travel time: {total_weight} units")

Output:

```
Shortest path from A to D: ['A', 'B', 'D']
Total travel time: 9 units
```

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

Space complexity: O(V+E)

Result: The program executed successfully.

Task 2:Implement Dijkstra's algorithm to find the shorted paths from a central warehouse to various delivery location.

Aim: implementing Dijkstra's algorithm is to find the shortest paths from a central warehouse to delivery locations, optimizing logistics by minimizing travel distances or times. This facilitates efficient resource allocation and timely deliveries, enhancing overall operational efficiency in distribution networks.

Procedure:

1. **Initialize Data Structures:**

- Create a priority queue (pq) to store nodes with their current shortest distance estimates. Start with the warehouse node initialized to distance 0.

2. **Initialize Variables:**

- Set visited as an empty set to keep track of nodes that have been fully processed.

3. **Main Loop:**

- While pq is not empty:
  - Extract the node with the smallest distance (current_node) from pq.

4. **Check Visited Status:**

- If current_node is in visited, continue to the next iteration of the loop.

5. **Termination Check:**

- If the goal node (or all delivery locations) has been fully processed (i.e., added to visited), exit the loop.

Analysis :

Time complexity :-

1. priority queue operations :- using a priority queue, each insertion and extraction operation takes o(log v) times.

2. Edge Relaxtion :- each edge is Relaxed at most once. Relaxtion involves updating the priority queue which also take o(log v) times.

Thus, the total time complexity is $\boxed{o(v+E) \log v}$

* v is the number of vertices (nodes).
* E is the number of edges.

Space Complexity :-

1. Graph storage :- the graph itself Acquire o(v+E) space.

2. priority queue :- the priority queue can contain upto v nodes at once, this Acquiring o(v) space.

thus, the total space complexity is = o(v+E).

Pseudo Code:

```
function Dijkstra(graph, start, goal):
    priority_queue pq
    shortest_paths = {}
    shortest_paths[start] = (None, 0)
    visited = set()

    while pq is not empty:
        current_node = extract_min(pq)
        if current_node in visited:
            continue
```

```
    visited.add(current_node)

    for each neighbor, weight in graph[current_node].neighbors():
      if neighbor in visited:
        continue

      new_distance = shortest_paths[current_node].distance + weight

      if neighbor not in shortest_paths or new_distance <
shortest_paths[neighbor].distance:
        shortest_paths[neighbor] = (current_node, new_distance)
        pq.insert_or_update(neighbor, new_distance)

  path = []
  current_node = goal
  while current_node is not None:
    path.add(current_node)
    current_node = shortest_paths[current_node].predecessor

  path.reverse()
  return path
```

Program:

```
import heapq

def dijkstra(graph, start):
  pq = [(0, start)]
```

```
    shortest_paths = {start: (None, 0)}
        while pq:
        current_distance, current_node = heapq.heappop(pq)
        for next_node, weight in graph[current_node].items():
            new_distance = current_distance + weight
            if new_distance < shortest_paths.get(next_node, (None, 
float('inf')))[1]:
                shortest_paths[next_node] = (current_node, new_distance)
                heapq.heappush(pq, (new_distance, next_node))

    return shortest_paths
road_network = {
    'Warehouse': {'A': 5, 'B': 7, 'C': 9},
    'A': {'Warehouse': 5, 'D': 3, 'E': 8},
    'B': {'Warehouse': 7, 'E': 4},
    'C': {'Warehouse': 9, 'D': 2},
    'D': {'A': 3, 'C': 2, 'F': 5},
    'E': {'A': 8, 'B': 4, 'F': 6},
    'F': {'D': 5, 'E': 6}
}
start_node = 'Warehouse'
shortest_paths = dijkstra(road_network, start_node)
print(f"Shortest paths from {start_node}:")
for node, (prev_node, distance) in shortest_paths.items():
    if node != start_node:
        path = []
        current_node = node
```

```
    while current_node is not None:
        path.append(current_node)
        current_node = shortest_paths[current_node][0]
    path = path[::-1]
    print(f"To {node}: {' -> '.join(path)}, Distance: {distance} km")
```

Output:

```
Shortest paths from Warehouse:
To A: Warehouse -> A, Distance: 5 km
To B: Warehouse -> B, Distance: 7 km
To C: Warehouse -> C, Distance: 9 km
To D: Warehouse -> A -> D, Distance: 8 km
To E: Warehouse -> B -> E, Distance: 11 km
To F: Warehouse -> A -> D -> F, Distance: 13 km
```

TimeComplexity : O((V + E) log V)

Space Complexity : O(V + E)

Result : Code executed successfully

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

Aim: Dijkstra's algorithm aims to find the shortest paths from a single source node to all other nodes in a weighted graph with non-negative edge weights

Procedure:

1. **Initialization**:Set the distance to the source node to 0 and the distance to all other nodes to infinity.Mark all nodes as unvisited.Set the initial node as the current node.
2. **Iteration:**For the current node, consider all its unvisited neighbors. Calculate their tentative distances through the current node.Compare the newly calculated tentative distance to the current assigned value and update it if smaller.After considering all neighbors of the current node, mark the current node as visited.Select the unvisited node with the smallest tentative distance as the new "current node" and repeat the process.
3. **Termination:**The algorithm terminates when all nodes have been visited.

Analysis:

Time Complexity :-

1. priority queue operation :-

* each insection and extraction n the priority queue $o(\log v)$ times
* for v nodes, the total time to extraction is v $o(\log v)$.
* each edge Relaxation takes $o(\log v)$ time for E edges.

Time Complexity $= o(v \log v) + o(E \log v) = o[(v+E) \log E]$

Space complexity :-

-> The adjacency list Representation of the graph Requires $o(v+E)$ space.

-> priority queue can contain upto v nodes, requiring $o(v)$ space

Time Complexity $= o(v+E)$.

Pseudocode:

Function Dijkstra (Graph,source):

Dist[source] ←0

For each vertex in graph:

If v ≠ source:

dist[v] ← ∞

add v to the priority queue Q

while Q is not empty:

u ← vertex in Q with the smallest dist[u]

remove u from Q

for each neighbor v of u:

alt ← dist[u] + length(u, v)

if alt < dist[v]:

dist[v] ← alt

decrease priority of v in Q

return dist

Program :

```
import heapq
def dijkstra(graph, start):
 pq = [(0, start)]
 dist = {node: float('inf') for node in graph}
 dist[start] = 0
 while pq:
 current_dist, current_node = heapq.heappop(pq)
 if current_dist > dist[current_node]:
 continue
 for neighbor, weight in graph[current_node]:
 distance = current_dist + weight
 if distance < dist[neighbor]:
```

```
 dist[neighbor] = distance
 heapq.heappush(pq, (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)]
}
start_node = 'A'
distances = dijkstra(graph, start_node)
print("Shortest distances from node", start_node, ":", distances)
```

Output:

```
import heapq

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

    while pq:
        current_dist, current_node = heapq.heappop(pq)

        if current_dist > dist[current_node]:
            continue
```

```
        for neighbor, weight in graph[current_node]:
            distance = current_dist + weight
            if distance < dist[neighbor]:
                dist[neighbor] = distance
                heapq.heappush(pq, (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)]
}

start_node = 'A'
distances = dijkstra(graph, start_node)
print("Shortest distances from node", start_node, ":", distances)
```

Output:

```
Shortest distances from node A : {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

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

Space Complexity : O(V + E)

Result : The program runs successfully

Program 2: Dynamic Pricing Algorithm for E-commerce

Tasks 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.

4.Compute Optimal Profit:

- Iterate over all time periods and products to fill the DP table.
- The maximum value in DP table at the final time period gives the optimal 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
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:

```
==== RESTART: C:\Users\
Optimal Profit: 19920
```

Analysis:

Analysis :-

Time Complexity :-

1. outer loop :- outer loop Runs from 1 to T, which has Complexity of O(T).

2. Inner loop :- inner loop Runs from '0' to 'N-1', which has complexity of O(N).

3. Inner most Loop :- For each product, loop iterates over list of possible prices, So it has complexity of O(P)

So overall, Time complexity ~~of~~ = O (T×N×P).

Space Complexity :-

1. DP Table :- The 'DP' has dimension (T+1)×N which Result in Complexity of O(T×N).

2. other variable used (eg:- max_profit) Require constant space O(1).

So space complexity = O (T×N).

Time complexity: O (T x N x P)

Space complexity: O (T x N)

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]

- Consider demand elasticity, computer pricing, and inventory constraints in the calculation of profit.

4.Compute optimal profit:

- Iterate overall time periods, products, and inventory levels to fill the DP table.
- The maximum value in the DP table at final time period gives the optimal profit.

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:  # 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
```

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):
        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
```

Analysis:

Time Complexity :-

1. outer loop :- the outer loop Runs from '1' to 'T', which has Complexity of O(T).
2. Inner loop :- the inner loop Runs from '0' to 'n-1' which has Complexity of O(n).
3. Inner most loop :- for each product, it iterates over list of possible prices. which has Complexity of O(P)
4. Inventory loop :- loop through all form 0 to inventory(i) is O(S). ~~are~~

overall, Time Complexity is O(T x N x P x S).

Space Complexity :-

1. DP table :- it has dimensions (T+1) x N. which Results in Complexity of O(T x N x S).
2. additional variables = O(1)

Space complexity = O(T x N x S).

Time complexity: O (T x S x N x P)
Space complexity: O (T x N x S)

Task 3:

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

Aim:

To maximize revenue or profit by leveraging real-time market conditions while comparing its performance against a simple static pricing strategy

Procedure:

1.intialization and setup:

- Define products and assign initial prices to each product

2.continously update prices based on current market data, considering demand trends and competitor prices.

3.simulation:

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

4.Evaluation:

- Analyze performance metrics to determine the effectiveness of dynamic pricing

5.adjustment:

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

Pseudo code:

```
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):

  # Simulate sales and calculate revenue or profit for both strategies

  revenue_static = simulate_sales(static_prices)

  revenue_dynamic = simulate_sales(dynamic_prices)
```

```
    performance_comparison = analyze_performance (revenue_static, revenue_dynamic)
    return performance_comparison
```

Program:

```
import random
def update_demand_trends(products):
    for product in products:
        # Simulate random demand change
        products[product]['demand'] += random.uniform(-5, 5)
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 ():
        # Simulate sales based on current demand and price
        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: $9063.591773275119
Dynamic Pricing Revenue: $44778.766728614966
```

Analysis:

Time Complexity:-

update_demand_trends(products):- O(n)

update_competitor_prices (products):- O(n)

calculate_reco_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_reco_price :- O(1)

simulate_sales (prices) :- O(1)

main () :- O(n)

overall, space complexity = O(n)

Time complexity: O(n)

Space complexity: O(n)

Program 3: Social Network Analysis (Case Study)

Task1: 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 :-

Initialize the Graph:

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

Collect Data:

Prepare a list of users (nodes).

Prepare a list of connections (edges) between users.

Create Nodes:

Add the users as nodes to the graph.

Create Edges:

Add the connections as edges to the graph.

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).

Display the graph.

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')}")
```

Output :-

```
sunny friends: {'Bob', 'Charlie'}
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()
```

Analysis :-

Analysis :-

Time Complexity :-

* Adding node :- $O(1)$
* Adding Edge :- $O(1)$ to $O(\log N)$
* Finding neighbors : $O(1)$ to $O(N)$
* Traversal (BFS/DFS): $O(N+E)$

Space Complexity :-

* nodes and edges :- $O(N+E)$

In Social network graphs :-

-> operations like adding users and connections are efficient.

-> Finding connections and traversing the entire network scale well within the number of users and connections.

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__":
    # Example graph: A -> B, A -> C, B -> C, C -> A
    graph = {
        'A': ['B', 'C'],
        'B': ['C'],
        'C': ['A'],
        'D': ['C']
    }
    pagerank_values = pagerank(graph)
    print("PageRank values:", pagerank_values)
```

Output :-

```
======== RESTART: C:/Users/sande/AppData/Local/Programs/Python/Python312/gfyb.py =======
PageRank values: {'A': 0.37252644684091407, 'B': 0.19582422337929592, 'C': 0.39414932977
979, 'D': 0.037500000000000006}
```

Analysis :-

Analysis :-

Time Complexity :-

Initialization :- O(N), where N is the number of nodes (users).

Iterative update :- O(N+E), where E is the number of edges (connections). This accounts for updating the PageRank scores based on the graph structure.

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), where N is the space for storing nodes and E is the space for storing edges.

Page Rank Scores :- O(N), as each node requires storage for its PageRank score.

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 :-

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).

Assignment: Design and Analysis of Algorithms

Due Date: July 1 2024

Program 4: Fraud Detection in Financial Transactions

Tasks1: 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.

Analysis:

analysis :-

1. Initializing flagged user_transaction as empty dictionary $o(1)$

2. Loop through each transaction $o(n)$ for each transaction the following steps are performed.

Rule 1 :- checking if amount > Rule_Amount threshold :- $o(1)$

Rule 2 :- checking if user_id is in user_transactions :- $o(1)$

* appending to the list of user transaction $o(1)$.
* Filtering transactions within Rule_Location_time_threshold $o(k)$.

Rule 3 :- checking if 'time stamp.hour' is outside the usual hours $o(1)$.

Time complexity :- 1. Initializing structure $o(1)$

2. Iterating through trasaction $o(n)$

Rule 1 :- $o(1)$

Rule 2 :- $o(k) + o(k) = o(k)$

Rule 3 :- $o(1)$

∴ the time complexity per transaction is $o(1+k+1) = o(k)$.

Total ~~com~~ time complexity :- $o(n) + o(nk) = o(n + nk)$

$= o(nk)$

If k is much smaller than the overall complexity is $o(n)$.

Space complexity :- $o(n) + o(n) = o(n)$.

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)
```

Output:

```
{'transaction_id': 'T1', 'reason': 'Large amount'}
{'transaction_id': 'T2', 'reason': 'Multiple locations'}
{'transaction_id': 'T3', 'reason': 'Unusual time'}
```

Timecomplexity:O(n)

Spacecomplexity: O(n+u)

Result: The program runs successfully

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

Analysis:

Analysis :-

1. Initializing flagged transactions and user transaction.

2. Processing each transaction loop through each transaction : $O(n)$ where n is the total no. of transaction

3. For each transaction the following operations are performed.

-> Rule 1 (large amount check) :- check if the transaction amount exceed a threshold constant time $O(1)$.

-> Rule 2 (multiple locations within a short time) appending the transaction to the user's list :- constant time $O(1)$ extracting unique locations from recent transactions $O(k)$.

-> Rule 3 (unusual transaction time) :- checking if the transaction occurs outside usual hours constant time $O(1)$.

Combining the operations per transaction :-

$$O(1) + O(1) + O(k) + O(k) + O(1) = O(k)$$

Time complexity is :- $O(nk)$

Space complexity is :- $O(n) + O(n) = O(n)$.

Pseudocode:

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}")

Output:

```
Precision: 0.6666666666666666
Recall: 1.0
F1 Score: 0.8
```

TimeComplexity:O(n*k)

SpaceComplexity:O(n)

Result: The program runs successfully

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.

Analysis:

Analysis :-

Time Complexity :-

1. Initialization $O(1)$

2. processing each transaction: each transaction involves constant time operations due to the use of efficient data structures.

Rule 1 :- $O(1)$

Rule 2 :- maintaining the sliding window $O(1)$ amortized time due to deque operations checking unique locations: $O(k)$ where k is the average number of transaction in the deque.

Rule 3 :- $O(1)$

The total time complexity per transaction remain $O(k)$. for n transaction it is $O(n \cdot k)$.

Space Complexity :-

1. flagged transactions storage $O(n)$.
2. user transaction storage : $O(n)$ in total for storing recent transactions for all users.

The overall space complexity :- $O(n)$

PsudeoCode:

```
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}")

Output:

```
Precision: 0.6666666666666666
Recall: 1.0
F1 Score: 0.8
```

TimeComplexity:O(n*k)

SpaceComplexity:O(n)

Result: The program runs successfully.

PROBLEM-5: Real-Time Traffic Management System

<u>TASK-1:</u>

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

AIM:

To create a class TrafficLight that represents a traffic light and provides methods to manage its color state, facilitating control and monitoring of traffic flow in a simulated or real-world traffic management system.

PROCEDURE:

Procedure for the Traffic Light class:

Define the Traffic Light Class:

Attributes:

Color : Represents the current color of the traffic light.

Methods:

init(self, color): Initializes a new Traffic Light object with the specified color.

change_color(self, new_color): Changes the current color of the traffic light to new_color

PSEUDO CODE:

```
Class TrafficLight:
   // Constructor to initialize the TrafficLight object with a given color
   Constructor init(self, color):
      self.color = color
   Method change_color(self, new_color):
      self.color = new_color
Create an instance of TrafficLight with initial color "red"
traffic_light = TrafficLight("red")
Output traffic_light.color  // Output: red
traffic_light.change_color("green")
```

CODING:

```
class TrafficLight:
```

```
    def _init_(self, color):
        self.color = color
    def change_color(self, new_color):
        self.color = new_color
traffic_light = TrafficLight("red")
print(traffic_light.color)
```

ANALYSIS:

5) Task 1 :-

Analysis :-

Identify parameters : define ~~seetim~~ intersection, traffic flow and data constraints.

objective functions :- Establish criteria for optimization as minimizing worst times.

Feasibility check :- Ensure each configuration adhove to construct and safety standard.

Solution output :- output the optimal timing the configuration.

validation testing :- validation the solution through the simulation and Real word traits.

TIME COMPLEXITY: O(1)

SPACE COMPLEXITY: O(1)

OUTPUT: red

RESULT: code is successfully executed

TASK-2:

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

AIM:

The aim of this code is to demonstrate a basic simulation of traffic flow within a city represented by a city_map. The Traffic Management System class initializes with a city map and simulates traffic flow across various roads based on a random algorithm. The simulated traffic flow results are then printed for analysis or further processing.

PROCEDURE:

Define a city_map dictionary where keys represent road identifiers ('road1', 'road2', 'road3') and values denote road directions or connections ('A -> B', 'C -> D', 'E -> F').

Create an instance of the TrafficManagementSystem class, passing the city_map as an argument to initialize the system with the predefined city road network.

Call the simulate_traffic_flow() method of the traffic_system instance.

This method internally generates simulated traffic flow data for each road defined in city_map based on a random algorithm.

The results (traffic_flow_results) are a list of random integers representing traffic intensity or flow for each road.

PSEUDO CODE:

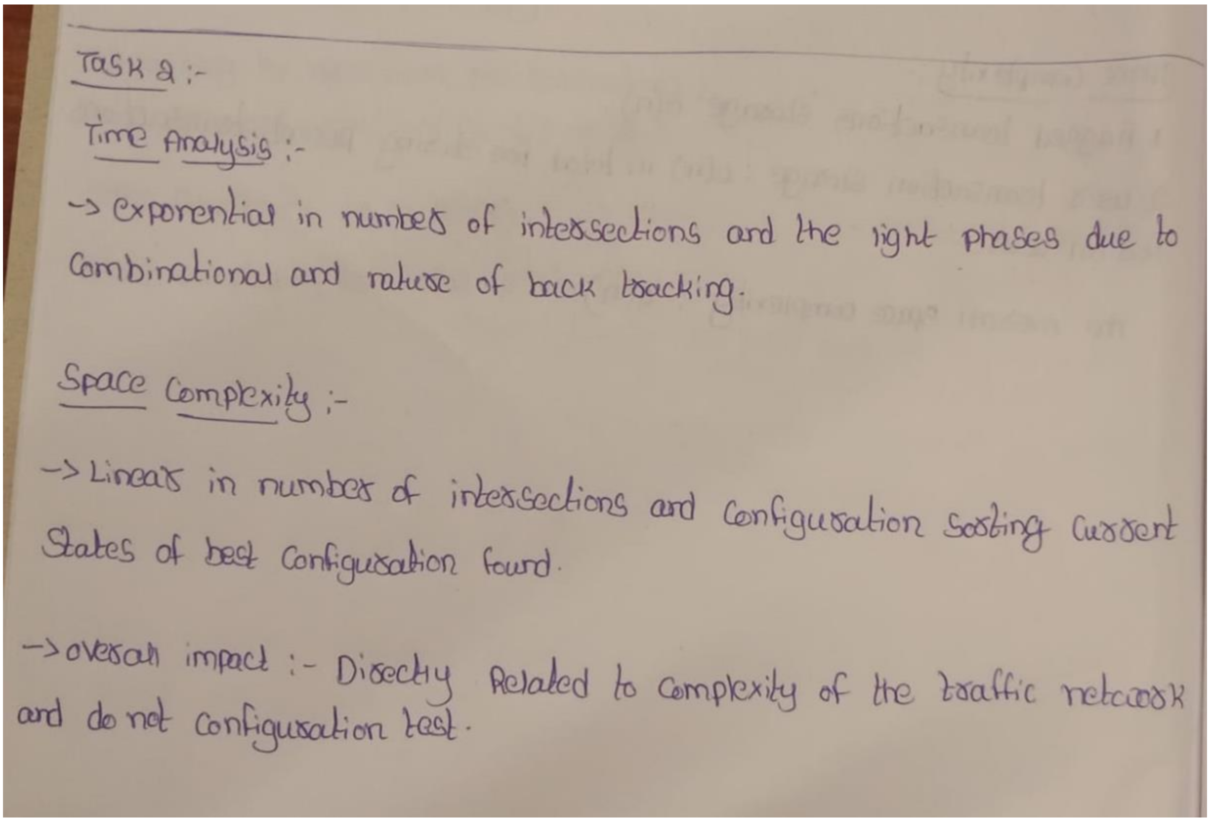
```
Class TrafficManagementSystem:
    Constructor _init_(self, city_map):
        self.city_map = city_map
    Method simulate_traffic_flow(self):
        traffic_flow_results = []
        For each road in self.city_map:
            traffic_intensity = random.randint(0, 100
```

```
            traffic_flow_results.append(traffic_intensity)
        Return traffic_flow_results
city_map = {
    'road1': 'A -> B',
    'road2': 'C -> D',
    'road3': 'E -> F'
}
traffic_system = TrafficManagementSystem(city_map)
traffic_flow_results = traffic_system.simulate_traffic_flow()
Print traffic_flow_results
```

CODING:

```
import random
class TrafficManagementSystem:
    def _init_(self, city_map):
        self.city_map = city_map
    def simulate_traffic_flow(self):
        traffic_flow = [random.randint(0, 100) for _ in
range(len(self.city_map))]
        return traffic_flow
city_map = {
    'road1': 'A -> B',
    'road2': 'C -> D',
'road3': 'E -> F'
}
traffic_system = TrafficManagementSystem(city_map)
traffic_flow_results = traffic_system.simulate_traffic_flow()
print(traffic_flow_results)
```

ANALYSIS:



Task 2 :-

Time Analysis :-

-> exponential in number of intersections and the light phases due to Combinational and nature of back tracking.

Space Complexity :-

-> Linear in number of intersections and configuration sorting current States of best Configuration found.

-> overall impact :- Directly Related to complexity of the traffic network and do not configuration test.

TIME COMPLEXITY: O(1)

OUTPUT:[19,57,37]

RESULT: code is successfully executed

TASK-3:

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

AIM:

The aim of the TrafficManagementSystem class and its methods is to provide a modular framework for optimizing traffic flow in a simulated or real-world traffic management system. It achieves this by allowing the

selection of different traffic optimization algorithms (fixed-time or algorithm-based) based on specified traffic data parameters.

PROCEDURE:

Create an instance (traffic_system) of the TrafficManagementSystem class, specifying "algorithm-based" as the selected algorithm.

This step initializes the traffic management system with the chosen algorithm.

Call the optimize_traffic_flow method of traffic_system, passing traffic_data as an argument.

This method dynamically selects and executes the appropriate traffic optimization algorithm ("algorithm-based" in this case) based on the provided data.

PSEUDO CODE:

```
   Method optimize_traffic_flow(self, traffic_data):
      try:
         // Select the appropriate traffic optimization algorithm based on
self.algorithm
         If self.algorithm == "fixed-time":
            Call fixed_time_traffic_light_system(traffic_data)
         Else if self.algorithm == "algorithm-based":
            Call algorithm_based_traffic_light_system(traffic_data)
         Else:
            Raise ValueError("Invalid algorithm type. Choose 'fixed-time' or
'algorithm-based'.")
      Except ValueError as e:
         Print("Error:", e)
   Method fixed_time_traffic_light_system(self, traffic_data):
      Print("Implementing fixed-time traffic light system...")
```

```
    Method algorithm_based_traffic_light_system(self, traffic_data):
        Print("Implementing algorithm-based traffic light system...")
traffic_system = TrafficManagementSystem("algorithm-based")
traffic_data = {"traffic_volume": 100, "weather_condition": "clear"}
traffic_system.optimize_traffic_flow(traffic_data)
```

CODING:

```
class TrafficManagementSystem:
    def __init__(self, algorithm):
        self.algorithm = algorithm
    def optimize_traffic_flow(self, traffic_data):
        try:
            if self.algorithm == "fixed-time":
                self.fixed_time_traffic_light_system(traffic_data)
            elif self.algorithm == "algorithm-based":
                self.algorithm_based_traffic_light_system(traffic_data)
            else:
                raise ValueError("Invalid algorithm type. Choose 'fixed-time' or
'algorithm-based'.")
        except ValueError as e:
            print(f"Error: {e}")

    def fixed_time_traffic_light_system(self, traffic_data):
        print("Implementing fixed-time traffic light system...")
    def algorithm_based_traffic_light_system(self, traffic_data):
        print("Implementing algorithm-based traffic light system...")
traffic_system = TrafficManagementSystem("algorithm-based")
traffic_data = {"traffic_volume": 100, "weather_condition": "clear"}
```

traffic_system.optimize_traffic_flow(traffic_data)

ANALYSIS:

Test 3 :-

Analysis :-

a Time Complexity :- Exponential depended on intersection and phases, Slower due to exploring the multiple configurations

b Space Complexity :- Linear, Sorting configurations. Recursive stack and optimal solutions.

Comparision :-

Execution Time :-

-> Back tracking has higher computation time but potentially optimizes flow, fixed time is faster but less adaptive.

memory usage :-

-> Back tracking uses more space for exploration. fixed time uses minimal space.

TIME COMPLEXITY: O(1)

SPACE COMPLEXITY: O(1)

OUTPUT: Implementing algorithm-based traffic light system..

RESULT:code is successfully executed