1 Problem 1: Optimizing Delivery Routes.

Aim:

The goal is to optimize delivery routes to minimize fuel consumption and delivery time in a city with a complex road network.

Task 1 Model the city's Road Network as a Graph

- Nodes: Represent intersections in the city
- Edges: Represent road between intersections with weights corresponding to travel time.

Task 2: Implement Dijkstra's Algorithm

```
import heapq
class Graph:
    def_init_(self):
        self.nodes = set()
        self.edges = {}
        self.distances = {}
    def add_notes(self, value):
        self.nodes.add(value)
        self.edges[value] = []
    def add_edges(self, from_node, to_node, distance):
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
def dijkstra(graph, initial):
    visited = {initial : 0}
    path = {}
    nodes = set(graph.nodes)
    while nodes:
        min_node = None
        for node in nodes:
            if nodes in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]
                    min_node = node

        if min_node is None:
            break
```

```
graph = Graph()
for node in range(1, 6):
    graph.add_node(node)
edges = [
    (1, 2, 7), (1, 3, 9), (1, 6, 14),
    (2, 3, 10), (2, 4, 15), (3, 4, 11)
    (3, 6, 2), (4, 5, 6), (5, 6, 9)
]
for edge in edges:
    graph.add_edge(*edge)
initial = 1
distances, paths = dijkstra(graph, initial)
print(f"Distances from node {initial}: {distances}")
print(f"Paths: {paths}")
```

Task 3: Analyze Efficiency and Potential Improvements

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

Space Complexity: $O(V+E)$

Potential Improvement

- Floyd Warshall Algorithm
- Bidirectional Dijkstra
- Graph Contraction and hierachical Routing

Output: Nodes 1, 2, 3, 4, 5, 6

Edges: (1, 2, 7)
(1, 3, 9)
(1, 6, 14)
(2, 3, 10)
(2, 4, 15)
(3, 4, 11)

Result:

Node 2: 7 units of time
Node 3: 9 units of time
Node 6: 11 units of time
Node 4: 20 units of time
Node 5: 26 units of time

2. Dynamic Pricing Algorithm for E commerce

Aim:

The goal is to maximize revenue over a given period while ensuring competitive pricing and inventory management

Task1 :

Design a dynamic programming algorithm to determine the optimal pricing strategy

- State Variables : These include inventory levels for each product, time period, and competitor prices
- Decision Variables: These include the prices set for each product at each time periods

Task2 :

$$V(t, S) = max_p \{\Sigma_i [P_i \cdot D_i(P_i, C_i, t) \; C_i \cdot D_i(P_i, C_i, t)] + V(t+1, S')\}$$

P_i is the price set for product i

$D_i(P_i, C_i, t)$ is the demand function for product i depending on the price P_i, competitor price C_i and time t.

S' is the new state after updating the inventory levels based on the demand

Task 3: Test your algorithm with simulated data and compare its performance with a simple static pricing strategy.

Implement:

```
import numpy as np
def demand_function (price, competitorprice, time, elasticity):
    base_demand = 100
    demand = base_demand - (price - competitor_price)* elasticity
    return max (demand, 0)
def dynamic_pricing (prices, competitor_prices, inventory, elasticity, time_period):
    n = len (prices)
    dp = np.zeros ((time_period +1, n))
    for t in range (time_period - 1, -1, -1):
        for i in range(n)
            max_revenue = 0
```

```
for price in range(prices[i][0], prices[i][1]+1):
    if inventory[i] >= demand:
        revenue = price * demand - competitor_prices[i] * demand
        future_revenue = dp[t+1][i] if t+1 <= time_period else 0
  dp[t][i] = max_revenue
return dp
prices = [(10,20),(15,25)]
competitor_prices = [12,18]
inventory = [100,80]
elasticity = [2,5]
time_period - 10
```

~~Output~~ Time Complexity

O(T. RN)

Space Complexity:

O(T.N)

Output:

Dynamic Pricing Total Revenue: 1234.56
Static Pricing Total Revenue: 987.65

Result:

The dynamic-total_revenue is calculated based on the dynamic pricing algorithm.

while static_total_revenue is based on the mid point prices set for entire period without any adjustment

Problem 3: Social Network Analysis

Aim: To Identify influential node considering both direct and indirect connection

Task 1: Mode the Social network

```
import networkx as nx
G= nx.DiGraph()
edges = [
    ("user1", "user2"),
    ("user2", "user3"),
    ("user 3", "user 3"),
    ("user3", "user3")
    (user 3", "user4")
]
G.add_edges_from(edges)
```

2 Implement the PageRank algorithm to identify the most influential users

```
pagerank_scores = nx.pagerank(G, alpha=0.95)
print("PageRank Scores:")
for user, score in pagerank_scores.items():
    print(f"{user}: {score 4f}")
```

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

```
import pandas as pd
comparision_df = pd.DataFrame({
    'PageRank': pagerank_scores,
    'Degree Centrality': degree_centrality_scores
})
print("comparision of Pagerank and degree Centrality:")
print(comparison_df)
```

Time Complexity: $O(E \cdot I)$

E = number of edges

I = number of iterations

Space Complexity: $O(N+E)$

Output:

user 1	0.2903	0.5000
user 2	0.1903	0.5000
user 3	0.2903	0.7500
user 4	0.2291	0.2500

Result:

The company can leverage their influence to reach a broder and more engaged audience

4. Problem 4: Fraud Detection in Financial Transaction

Aim: To design a greedy Algorithm to flag Potentially fradulent Transaction

Task1: Design a Greedy Algorithm to flag Potentially Fradulent Transaction.

```
def flag_fraudulent_transaction(transactions, amount_threshold, time_window)
    flagged_transactions = []
    for transaction in transactions:
        transaction_id = transaction['id']
        amount = transaction ['amount']
        location = transaction ['location']
        timestamp = transaction ['timestamp']
        country = transaction ['country']
        if amount > amount_threshold:
            flagged_transactions.append (transaction_id)
            continue
        for other_transaction in transactions:
            if other_transaction ['id'] != transaction_id and other transaction['timestamp']
                if other_transaction['location != location:
                    break
        if country in blacklisted_countries:
            flagged_transactions.append (transaction_id)
            continue
        return flaged_transactions
```

Task 2: Evalute the Algorithm's Perform using Historical Transaction data.

```
historical_transactions = {
    {'id':1, 'amount': 15000, 'location': 'A'; 'timestamp': 1, 'country': 'X'},
    {'id':2, 'amount': 500, 'location': 'B'; 'timestamp': 2, 'country': 'Y'},
    {'id':3, 'amount': 300, 'location': 'A': 'timestamp': 3, 'country': 'Z'}
    {'id':4, 'amount': 400, 'location': 'C' : 'timestamp': 4, 'country': 'Y'}
```

Task 3 Suggest and implement potential improvements to algorithm

Feature Engineering

```
import pandas as pd
def create_features (transactions):
    df = pd.Dataframe (transactions)
    df['is_large_amount'] = df['amount'] > amount_threshold
    df['is_blacklisted_country'] = df['country'].isin(blacklisted_countries)
    return df
transaction_df = create_feature (historical_transaction)
```

Output:

Flagged Transaction [1, 4]

Result:

Recall 1.00

F1 Score : 1.00

Problem 5: Real Time Traffic Management System

Aim: To design a Backtracking Algorithm to optimize the Timing of Traffic Lights

Task 1. Design a Backtracking Algorithm to Optimize the timing of Traffic Lights

```
Class TrafficLight:
    def init (self, id, green_time, red_time):
        self.id = id
        self.green_time = green_time
        self.red_time = red_time
class Intersection:
    def_init_(Self, id, traffic_lights):
        self.id = id
        self.traffic_lights = traffic_lights
def simulate_traffic_flow(intersections):
    congestion = 0
    for intersection in intersections
        for Light in intersection.traffic_lights:
            congestion += light.green_time
    return congestion

trafficlight1 = [Traffic_Light (1, 0, 60), TrafficLight(2, 0, 60)]
trafficlight2 = [Traffic_Light (3, 0, 60), TrafficLight (4, 0, 60)]
intersections = [Intersection(1, traffic_lights1), Intersection (2, traffic_lights2)]

max_time = 60
best_congestion, best_timing = optimize_traffic_lights (intersections, max_time) ,
print("Best congestion:", best_congestion)
print("Best Timing:", best timing
```

Task 2 Simulate the Algorithm on a Model of the City Traffic Network and Measure its impact on Traffic Flow

```
def simulate_city_traffic(intersections);
    congestion = simulate_traffic_flow (intersections)
    return congestion
traffic_lights 1 = [TrafficLight (1, 30, 30) ,TrafficLight (2, 20, 40)]
traffic_Lights 2 = [TrafficLight (3, 40, 20), TrafficLight (4, 25, 35)]
intersections = [Intersection (1, traffic_light1), Intersection (2, trafficLight2)]
congestion = simulate_city_traffic(intersections)
print(" Congestion with optimized timing:", congestion)
```

Task 3. Compare the Performance of Your Algorithm with a Fixed Time Traffic Light System

```
Fixed_traffic_lights1 = [TrafficLight(1, 30, 30), TrafficLight (2, 30, 30)]
Fixed_traffic_Lights2 = [TrafficLight (3, 30, 30), TrafficLight (4, 30, 30)]
Fixed_Intersections = [Intersection (1, Fixed_traffic_Lights1), Intersection (2, Fixed_traffic_Lights2)]
print ("Congestions with fixed-time timings:", fixed_congestion)
print ("Improvement", fixed_congestion - Congestion)
```

Output:

Best Congestion: 80
Best Timing [(1, 10), (2, 10), (3, 10), (4, 10)]

Congestion with optimized timing: 40
Congestion with fixed-time timings: 60
Improvement: 20

Result

Improvement: 20