Problem 1:- Optimizing delivery Routes (case study)

Aim:-
To optimize delivery routes for a logistics company by minimizing fuel consumption and delivery time using Dijkstra's algorithm on a graph model of a city's road network

Procedure:-

1. Model the city's road network:
   - Represent intersection as node
   - Represent roads as edges with weights indicating travel time.

2. Implement Dijkstra's Algorithm:
   - Use priority queue to efficiently retrieve the next node with the smallest tenative distance.
   - Update distances to neighboring nodes and keep track of the shortest paths.

3. Analyze the algorithm's Efficiency:
   - Evaluate the time and space complexity
   - Discuss potential improvements or alternative g algorithms.

4. Graph Model of the city's road network
   - Nodes: Represents intersection
   - Edges: Represent road, with weights as travel times.

Example graph representation:-

A --5--> B

A --3--> C

B --2--> D

C --2--> D

D --1--> E

Pseudo code for Dijikstra's Algorithm

```
function dijkstra(graph, source):
        dist[source] ← 0
        create priority queue Q
        for each node v in ~~queue Q~~ graph:
            if v ≠ source:
                dist[v] ← ∞
            Q.add_with_priority(v, dist[v])
      while Q is not empty:
          u ← Q.extract_min()
          for each neighbor v of u:
              alt ← dist[u] + weight(u,v)
              if alt < dist[v]:
                  dist[v] ← alt
                  Q.decrease_priority(v, alt)

      return dist
```

Coding Analysis

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

    while pq:
    curr_dist, curr_node = heapq.heappop(pq)
    if curr_dist > distances[curr_node]:
        continue

    for neighbor, weight in graph[curr_node].item():
        distance = curr_dis + weight
        if distance < distances[neighbor]:
            distances[neighbor] = distancer
            heapq.heappush(pq, (distance, neighbor))

return distances

graph = {
  'A': {'B': 5, 'C': 3},
  'B': {'D': 2},
  'C': {'D': 2};
  'C': {'E': 1}
  'E': {}
}
Start = 'A'
```

```
distances = dijkstra(graph, start)
print(distances)
```

✲ Output & Result

{'A': 0, 'B': 5, 'C': 3, 'D': 5, 'E': 6}

- Shortest path from 'A' to 'B' is 5, from 'A' to 'C' is 3, and so on.

✲ Time Complexity

$T(n) = O((V+E) \log V)$

- V => no. of vertices
- E => no. of Edges

✲ Space Complexity

O(V) , for storing the distances and priority queue.

✲ Reasoning

- **Suitability of Dijkstra's Algorithm:**
  - Dijkstras algorithm is suitable for finding the shortest path in graph with non-negative weights, which alligns with our road-network model where travel times are non-negative.

- Assumptions:
  - The weights (Travel times) are non-negative
  - The graph is connected

- Road Conditions:
  - Traffic and road closures can be modeled by dynamically adjusting edge weights or removing edges, respectively.
  - Real-time data can be incorporated to update the graph and recompute routes as needed.

- Alternate algorithm:
  - Bellman Ford Algorithm :- Suitable if there are negative weights, though it has higher time Complexity $O(VE)$

Problem 2: Dynamic Pricing algorithm for E-commerce

Aim :- To design and implement a dynamic pricing algorithm that optimizes the prises of products in real-time based on demand and competitor prices, using dynamic programming.

Procedure :-

1. Design Dynamic Programming Algorithm :-
   - Define state variables representing inventory levels, time periods, and pricing ~~strag~~ strategies.
   - Create a recursive relation to maximize revenue based on these states.

2. Incorporate Factors :-
   - Include inventory levels, competitor pricing, and demand elasticity in the state transitions and revenue calculations.

3. Test Algorithm with simulated data :-
   - Compare the performance of dynamic pricing algorithm against a simple static pricing strategy using simulated data.

CamScanner

✱ Pseudocode for Dynamic Pricing Algorithm :-

```
function Dynamic_pricing(products, periods, inventory, demand, comptitor-
                          price, elasticity):
    dp = array with dimensions(products, periods, inventory) initialized
                                                              to 0
    for p in range(products):
        for t in range(periods):
            for i in range (inventory[p] + 1):
                for price in possible_price:
                    expected_d = demand[p][t] * elasticity[p][price] *
                                      competitor_price[p][t]
                    actual_demand = min(expected_demand, i)
                    revenue = price * actual_demand
                    dp[P][t][i] = max(dp[p][t][i], revenue + dp[p]
                                          [t-1][i-actual_demand])
    optimal_pricing = []
    for p in range(products):
        optimal_price = []
        for t in range(periods):
            optimal_price.append(find_max_price(dp[p][t]))
        optimal_pricing.append(optimal_price)
    return optimal_pricing.
```

Coding Analysis :-

```
import numpy as np
def dyna-price(pro, per, inven, demand, comp-p, elasticity):
    dp=np.zeros((len(pro), per, max(inven)+1))
    possible-price = np.linspace (10, 100, 10)
    for p in range(len(products)):
        for t in range (per):
            for i in range(inven[p]+1):
                for price in possible-prices:
                exp-d = demand[p][t] *elasticity[p](price,
                                        comp-p[p][t])
                actual-demand = min(exp-d, i)
                revenue = price * actual-demand
                if t>0 and i-actual-demand >=0:
                    dp[p][t][i] = max(dp[p][t][i], revenue +
                                    dp[p][t-1][i-actual-demand])
            else:
                dp[p][t][i] = max(d[p][t][i], revenue)
 optimal pricing = []
 for p in range (len(products)):
    optimal-price = []
    for t in range (per):
        max-price= possible-price[np.argmax(dp[p][t])]
        optimal-price.append(max-price)
    optimal-pricing.append(optimal-price)
 return optimal-pricing
```


CamScanner

```
Products = ['A', 'B']
periods = 10
inventory = [100, 80]
demand = [np.random.rand(periods) for _ in products]
comp-price = [np.random.rand (periods) * 100 for _ in products]
elasticity = [lambda price, comp-price : 1-price for_in products]
optimal-pricing = dyna-price (products, periods, Inventory, demand,
                              comp-price, elasticity)
print (optimal-pricing)
```

Analysis of Benefits and drawbacks of Dynamic Pricing :-

- **Benefits:**

1) Maximized Revenue; by adjusting prices in real-time
2) Inventory management; Dynamic pricing helps balance inventory levels and avoid stock outs or excess stock.
3) Competitive Edge; By reacting to competitor prices, the company can remain competitive.

- ~~Benefits~~.

- **Drawbacks :-**

1) Complexity; implementing and maintaining a dynamic pricing algorithm requires significant computational resources and data.
2) Customer perception: Frequent price change may lead to customer dissatisfaction.
3) Market Sensitivity : Overly aggressive pricing strategies might trigger price wars.

Reasoning :-

- Dynamic Programming Justification: Dynamic programming is suitable for this program as it breaks down the decision making process overtime and Inventory levels into smaller, mannagable subproblems, ensuring optimal decisions at each steps.

- Factors Incorporated:-
  - Inventory levels: Ensure price are set to avoid stockouts.
  - Competitor Pricing: Adjust price based on competitor data.
  - Demand Elasticity:- incorporate the sensitivity of demand to price changes.

- Challenges:-
  - Data Accuracy
  - Computational accuracy
  - Customer reactions

Problem 3 :- Social Network Analysis (Case Study)

Aim:- To Identify influential users in a social network by implementing and comparing the pageRank algorithm and degree centrality

Procedure

1. Model the Social network
   - Represent users as nodes and connections as edges in a graph.

2. Implement page rank algorithm
   - Calculate the page rank score for each node to measure influence

3. Compare with degree centrality
   - Calculate the degree centrality for each node
   - Compare the results with Page Rank scores.

4. Graph model of Social network
   - Nodes : represent users
   - Edges ; represent connections between users

Example graph representation:-

A --> B
A --> C
B --> C
C --> A
D --> C
E --> F

✡ Pseudocode for Page rank algorithm :-

```
function pagerank (graph, d=0.85, max_inter=100, total=1.0e-6):
    N= no. of nodes in graph
    rank = array of N elements intialized to 0
    for i from 1 to max iterations:
        for each node u in graph:
            new_rank[u] = (1-d)/N
            for each node v linking to u:
                new_rank[u]+=d* rank[v]/no. of outgoing links
                                                from v
        if sum of absolute differences between new_rank
                                        and rank < tol:
            break
        rank= new_rank
    return rank
```

✡ Coding Analysis :-

```
def pagerank(graph, d=0.85, max_iter=100, tol= 1.0e-6):
    N = len(graph)
    rank= np.ones(N)/N
    new_rank = np.zeros(N)
    adjacency_Matrix = np.zeros((N, N))
    for u in range(N):
        for v in graph[u]:
            adjacency_matrix[u][v] = 1
    outdegree = np.sum(adjacency_matrix, axis = 1)
    for i in range(max_iter):
        for u in range(N):
            new_rank[u] = (1-d) /N
            for v in range(N):
```

```
        if adjacency_matrix[v][u] == 1:
            new_rank += d * rank[v] / outdegree[v]

    if np.linalg.norm(new_rank - rank, 1) < tol:
        break

    rank = new_rank.copy()

  return rank

graph = [
    [1, 2],    # A -> B -> A D -> C
    [2],       # B -> C
    [0],       # C -> A
    [2],       # D -> C
    [ ]        # E (no connections)
    [ ]        # F (no connections)
]
pagerank_score = pagerank(graph)
print(pagerank_score)
```

Degree Centrality Measure :-

Degree centrality is simply the measure or count of the number of edges connected to a node.

```
def degree_cen(graph):
    N = len(graph)
    in_degree = np.zeros(N)
    out_degree = np.zeros(N)
    for u in range(N):
        out_degree[u] = len(graph[u])
        for v in graph[u]:
            in_degree[v] += 1
    return in degree.
in_deg, out_deg = degree_cen(graph)
```

print ("Indegree Centrality: ", In_deg)
print ("Out-degree Centrality : ", out_deg)

Comparison of Page Rank and degree Centrality Results:

print ("Page Rank scores: " page_rank_scores)
print ("In-degree centrality: " in_deg)
print ("Out-degree centrality: "out_deg)

Reasoning :-

- Page Rank effectiveness :
  - Page rank is effective ~~but~~ because it not only consider the no. of connection, but also the quality of these connection. Nodes ~~to~~ that are linked by other important nodes gains higher scores.
  - This is crucial in identifying influential users as it reflects the overall connectivity and importance ~~of~~ within the network

Conclusion:-

By modelling the social network as a graph and implementing the page rank algorithm, we can effectively identify influencial users. Comparing page rank algorithm, we can effectively, with degree centrality highlight the additional insights provided by considering the quality of connections, making page rank a more robust measure in many ~~sc~~ scenarios.

Problem 4 : Fraud detection in Financial Transactions

Aim :- To develop a greedy algorithm to detect ~~pott~~ potentially fradulant transactions in real-time based on predefined rules and evaluate its performance using historical transaction data

Procedure :-

1. Design a greedy algorithm
   - Define a set of rules to identify suspicious transactions.
   - Implement the algorithm to flag transactions that violate these rules
2. Evaluate Algorithm Performance :
   - Use historical transaction data to assess the algorithm's performance
   - Calculate precision, recall, and F1 score.
3. Suggest and implement Improvements :
   - Analyse the algorithm's limitation and propose enhancements
   - Implement the suggested improvements and evaluate their impact.

Pseudocode ~~fors~~ Fraud detection Algorithm

```
function detect_fraud (transaction, rules):
    flagged_trans = []
    for trans in transactions:
        for rule in rules:
            if rule.is_violated (transaction):
                flagged_transactions.append (transaction)
                break
```

return flagged_trans.

Coding Analysis :-

```
Class Transactions:
    def _init_(self, user_id, amount, location, timestamp):
        self.user_id = user_id
        self.amount = amount
        self.location = location
        self.timestamp = timestamp.
        self.timestamp = timestamp.
def detect_f(transaction, rules, recent_t_timewin):
    flagged_tran = []
    for transaction in transactions:
        for rule in rules:
            if rule(transaction, recent_t, time_win):
                flagged_trans.append(transaction)
                break.
    return flagged_transaction
```

Performance Evaluation using Historical data :-

- Evaluation meteler :-
- Precision : The proportion of correctly identified fradulent transaction out of flagged transaction
- Recall : The proportion of correctly identified fradulent transaction out of all actual fradulent transaction.
- F_1 score : The harmonic mean of precision and recall.

Reasoning :-

- Greedy algorithm sustainability:
  - Real time fraud detection requires immediate responses. A greedy algorithm is efficient in flagging transactions based on predefined rules without complex computation.

- Trade offs:
  - Speed: The greedy approach ensures fast detection
  - Accuracy: While the greedy method is fast, it might miss complex patterns of fraud.

Problem 5:- Real Time Traffic management system

Aim:-
To design a back tracking algorithm to optimize the timing of traffic lights at major Intersections.

Procedure:-

1. Design a back tracking algorithm:-
   - Define the state and constraints for traffic light timing optimization
   - Implement the back tracking algorithm to explore possible timing configuration
2. Simulate the Algorithm:-
   - Create a model of city's traffic network
   - Simulate the back tracking algorithm
3. Compare performance:-
   - Compare the optimized traffic light system with a fixed-time traffic light system in terms of congestion reduction. and overall traffic flow.

Pseudocode for Traffic light optimization

```
function optimize_traffic (inter, max_green, traffic_flow):
    best_config = None.
    best_traffic_flow = infinity
    def back_track (curr_inter, curr_config):
        if curr_inter == len (inter) :
            current_flow = simulate_traffic
```


CamScanner

```
    if curr_flow < best_traffic_flow:
        best_traffic_flow = curr_flow
        best_config = current_config.copy()
    return
for green_t in range(1, max_green_time + 1):
    curr_config [curr_inter] = green_time
    backtrack (curr_inter + 1, curr_config)
    curr_config [current_inter] = 0

backtrack (0, [0] * len(inter))
return best_config, best_traffic_flow
```

Simulation Result and Performance Analysis :-

```
def fixed_time(inter, fixed_g, traffic_mat):
    return simulate_traffic ([fixed_g] * len(inter), traffic_mat)

fixed_g = 5
fixed_flow = fixedtime(inter, fixed_g, traffic_mat)
```

Reasoning :-

- Justification for using back tracking
    - Exploration of state space
    - Constraints handling.
- Complexities in real-time traffic management:
    - Dynamic traffic pattern
    - Multiple constraints.
    - Scalability.