① Optimizing Delivery Routes

Graph Model of the city's Road Network

* Nodes : Intersections
* Edges : Roads
* Weights : Travel time

Pseudocode & Implementation of Dijkstra's Algorithm

```
function Dijkstra (graph, start):
    dist = {node: inf for node in graph}
    dist [start] = 0
    Priority_queue = [(0, start)]
    while priority_queue is not empty:
        current_dist, current_node = heappop (Priority_queue)
        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
                heappush (priority_queue, (distance, neighbor))

return dist
```

Analysis of Algorithm

Efficiency: $O(V \log V + E)$ where $V \rightarrow$ vertices
$E \rightarrow$ edges

Improvements:

* Algorithm incorporates heuristics to speed up the search.
* Floyd Warshall Algorithm: For finding shortest paths between all pairs of nodes.
* Traffic & road conditions: Dynamically update weights based on real-time traffic data and road closures.

Reasoning:

* Suitability: Dijkstra's Algorithm is suitable for finding the shortest path in a graph and non negative weights, representing travel times.
* Assumptions: Travel times (weights) are non-negative.
* Road conditions: changes in traffic and road closures can be managed by dynamically updating the graph's weights to reflect real time conditions.

② Dynamic Pricing Algorithm for E-commerce

Pseudocode & Implementation of Dynamic Pricing Algorithm:

```
function dynamic-pricing (prices, demand, competitor-prices, inventory-time-period):
    n = len (prices)
    dp = [[0] * (inventory+1) for - in range (time-period+1)]
    for t in range (1, time-Period+1):
        for inv in range (inventory+1):
            max-Profit = 0
        for p in range (n):
            if inv > 0:
                competitor-price = competitor-prices [t-1] [p]
                adjusted-Prices = Prices [p] - (Prices [p] - competitor-price) * demand [t-1] [p]
                Profit = adjusted-Price * demand [t-1] [p] + dp [t-1] [inv-1]
                max-Profit = max (max-Profit, profit)
        dp [t] [inv] = max-Profit
return dp [time-Period] [inventory]
```

Simulation results comparing Dynamic & Static Pricing strategies

* **Static Pricing Strategy:** Fixed prices for products regardless of demand and competitor prices.

* **Dynamic Pricing Strategy:** Adjust prices based on demand, competitor prices and inventory levels.

Analysis:

Benefits:

→ Maximizes revenue by adjusting prices in real-time based on demand and competition.

→ Responds to market conditions, leading to better inventory management and customer satisfaction.

Drawbacks:

→ Complexity in implementation and maintenance.

→ Requires accurate & real-time data for optimal performance.

→ Potential customer dissatisfaction due to frequent price changes.

Reasoning:

Justification for Dynamic Programming: Suitable for optimizing Pricing strategies over a given period, taking into account multiple factors and their interdependies.

Incorporating different factors:

* Demand Elasticity: Adjust prices based on demand levels
* Competitor Pricing: Respond to competitor prices to remain competitive.
* Inventory levels: Ensure inventory is managed effectively to avoid stockouts or overstock.

Challenges Faced:

* Data Availability: Ensuring real time and accurate data for demand and competitor prices.
* Algorithm complexity: Balancing between optimal pricing and computational efficiency.

③ Social Network Analyses (Case Study)

Graph Model of the Social Network

* Nodes: users
* Edges: connections between users

Pseudocode & Implementation of PageRank Algorithm:

```
function PageRank (graph, d = 0.85, max_iter = 100, tol = 1e-6):
  N = len (graph)
  ranks = {node: 1/N for node in graph}
  for i in range (max_iter):
    new_ranks = {}
    for node in graph:
      rank_sum = 0
      for neighbors in graph:
        if node in graph [neighbors]:
          rank_sum += ranks [neighbors] / len (graph [neighbors])
return ranks
```

Comparison of Degree centrality results and PageRank

Definition: No. of connections a user has.

Reasoning:

Effectiveness: measures influence based on quality & quantity, not just no. of connections. It considers the importance of each connection, making it a more robust measure for identifying influential users.

Preference in Different Scenarios:

* Useful in scenarios where connections are of equal importance – Degree centrality
* Preferred in scenarios where quality of connects matters, such as social media networks.

4. Fraud Detection in Financial Transactions

Pseudocode & Implementation of the Fraud Detection Algorithm

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

Suggestions & Implementation of Improvements

* Anomaly Detection : Implement machine Learning Models
* Feature Engineering : Add more features such as transaction frequency.
* Hybrid Model : Combine rule based & machine Learning approaches.

Reasoning :

Suitability of Greedy Algorithm : Suitable for real-time fraud detection as it quickly evaluates transactions based on pre-defined rules, ensuring immediate response without complex computations.

Trade-offs between Speed & Accuracy :

* Speed : Greedy approach is fast but may miss complex fraud patterns.
* Accuracy : Anomaly detection can improve accuracy.

Addressing Trade-offs :

* Greedy algorithm addresses speed by evaluating simple rules.
* Adding machine Learning models can (enhance) balance speed and accuracy.

5. Real-time Traffic Management System:

Pseudocode & Implementation of Traffic Light optimization Algorithm

```
function optimize_traffic_lights (intersections, max_time):
    best_timing = None
    best_traffic_flow = float('inf')
    for time in range(1, max_time+1):
        current_timing[intersection_index] = time
        backtrack(current_timing, intersection_index+1)
backtrack([0]*len(intersections), 0)
return best_timing
```

Comparison with a Fixed-Time Traffic Light System:

Optimized System:

* Adjusts timings dynamically based on traffic flow simulation.
* Reduces congestion & improves traffic flow efficiency.

Fixed-Time System:

* uses constant timing for traffic lights regardless of traffic conditions.
* Simple but can lead to inefficiencies during Peak and off-peak hours.

Results:

* The optimized timing showed a significant reduction in traffic congestion compared to fixed-time system, demonstrating the effectiveness of the backtracking approach in adjusting timings based on real-time data.

Reasoning:

Justification for Backtracking:

* Backtracking is suitable for exploring all possible combinations of traffic light (condition) timings to find the optimal solution.
* It ensures that all potential configurations are considered, leading to optimized traffic flow.

Complexities in Real-time Traffic Management

* Data variability: Traffic patterns can vary significantly throughout the day.
* Computational Resources: Real-time optimization requires efficient algorithms to provide timely adjustments.
* Coordination: Synchronizing traffic lights accross multiple intersections adds complexity.

Addressing Complexities:

The Algorithm uses backtracking to explore configurations and simulations to measure impact, ensuring a balanced approach to optimize traffic flow efficiently.