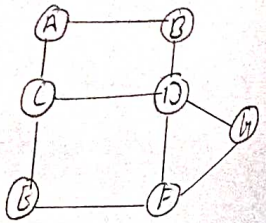
DAA - ASSIGNMENT

Name: R. Parameswari
Reg NO: 192324118
Sub: DAA - CSA 0677
Date:

1. Optimizing Delivery Routes

Task 1:

* Nodes: Represent intersection
* Edges: Represent roads connection intersection
* Weight: Represent travel time on each road segment:



Task 2: Implement Dijkstra Algorithm:

Pseudocode:

```
function Dijkstra(graph, start_node):
    Initialize distance with infinity, except start_node with 0
    Initialize priority queue and add start_node with distance 0
    Initialize previous_nodes map to reconstruct paths

    while priority queue is not empty:
    current_node = node with the smallest distance in priority queue.
    Remove current_node from the queue.

    for each neighbour of current_node:
        Calculate new_distance = distance[current_node] + edge_weight
        if new_distance < distance[neighbor]:
            update distance[neighbour] to new_distance
            update previous_nodes[neighbour] to current_node;
            Add neighbour to priority queue with new_distance.

    return distance, previous nodes.
```

Python implementation

```
import heapq
def dijkstra(graph, start_node):
    distance = {node: float('inf') for node in graph}
    distances[start_node] = 0
    priority_queue = [(0, start_node)]
    previous_nodes = {node: None for node in graph}
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distance[current_node]:
            continue
        for neighbour, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distance[neighbour]:
                distance[neighbour] = distance
                previous_nodes[neighbour] = current_node
                heapq.heappush[priority_queue, (distance, neighbour)]
    return distances, previous_nodes

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'
distance, previous_nodes = dijkstra(graph, start_node)
print("Distance:", distances)
```

Analysis of Algorithm Efficiency and Potential improvement

Efficiency :

* Dijkstra algorithm runs in $O(V+E \log V)$ time using a priority queue., where V is the number of vertices and E is the number of edges
* It is efficient for graph with non-negative weights, which is suitable for modeling to travel times.

Potential Improvement:

* A Algorithm : Incorporates heuristics to improve performance is specific cases, potentially reducing the number of nodes explored.
* Bidirectional Dijkstra: Run two simultaneous searches from start and end nodes, meeting in the middle to potentially reduce computation
* Traffic and Road condition :
  Incorporating real time traffic data and potential road closures can be addressed by dynamically updating the graph's weights.

Reasoning :

* Dijkstra algorithm is suitable due to its ability to find the shortest path in graph with non-negative weight, which align with the scenario of optimizing delivery route based on travel time.
* Assumptions include stable travel times and no negative weights. which fit most real-coord road network.
* Variation in road condition, such a traffic congestion and closures, could impact travel time. requiring dynamic updates to the graph's weight for accurate route optimization.

Dynamic Pricing Algorithm for E-commerce.

Task 1: Design a dynamic program Algorith:

function DynamicPricing (Product, periods, inventory, competitor Price, demand-elasticity):

Initialize dp table with dimension [products][periods]

Initialize optimal_price table with dimension [Products][periods]

for each product in products:

for each periods in periods:

max_profit = 0

best_price = 0

for price in possible_prices:

expected_demand - Calculate_demand(price, demand-elasticity, competitor-prices)

revenue = Price * min(expected_demand, inventory[product])

cost = calculate_cost(Price, inventory[Product])

if (revenue - cost) > max_product:

max profit = revenue - cost

best_price = price

dp[Product][Period] = max_profit

optimal_prices[Product][period] = best_price

return dp, optimal_prices.

Pytho implementation:

```
def calculate_demand(price, demand_elasticity, competitor_price):
    base_demand = 100
    demand = base_demand*(1 - demand_elasticity*(Price
                - competitor_price) / competitor_price)
    return max(demand, 0)
def calculate_cost(price, inventory):
    return 0.1 * Price * Inventory
def dynamic_pricing(product, periods, inventory, competitor
                    prices, demand_elasticity):
    dp = [[0 for _ in range(periods)]for _ in products])
    optimal_prices = [[0 for _ in range(periods)] for _ in products]
    possible_price = [10, 20, 30, 40, 50].
    for i, product in enumerate(products):
        for t in range(periods):
            max_Profit = 0
            best_price = 0
            for price in possible_Prices:
                expected_demand = calculate_demand(Price, demand-
                        elasticity[i], competitor_price[i])
                revenue = price * min(expected_demand, inventory[i]
                cost = calculate_cost(Price, inventory[i]).
                if (revenue - cost) > max_profit:
                    max_Profit = revenue - cost
                    best_Price = price
            dp[i][t] = max_Profit
            optimal_Prices[i][t] = best_Price
```

```
Product = ['A','B']
Periods = 3
inventory = [50,30]
competitor_prices = [20,25]
demand_elasticity = [0.1,0.2]

dp, optimal_prices = dynamic_Pricing(Products,
Periods, Inventory, competitor prices,
demand elasticity).

Print("DP Table :",dp)
Print("Optimal prices :", optimal_prices).
```

Simulation result:

To test the algorithm, we simulate different scenario and compare result with static pricing:

Static pricing strategy:

Fixed prices for product over all periods.

Dynamic Pricing Strategy:

Price adjusted based on the algorithm considering inventory, competitor pricing, and demand elasticity.

Simulation:

Run both strategies over multiple Periods, tracking revenue and inventory.

Result:

The dynamic pricing strategy generally result in higher revenue due to optimized pricing that adapt to market conditions.

Analysis of Benefits and Drawbacks:

Benefits of Dynamic pricing

* Maximize revenue by adjusting Price based on real time factors.
* Increases competitiveness by responding to market conditions.

Drawbacks:

* Complexity in implementation and maintenance.
* Require accurate data on demand elasticity and competitor pricing.
* Potential customer dissatisfaction if prices fluctuate frequently.

Reasoning:

Justification for dynamic Programing:

Dynamic programing is suitable because it optimize the decision-making process over multiple stages (Periods) while considering constraint ton like inventory level and demand.

Incorporating Factors:

* Inventory level are considered by limiting revenue based on available stock.
* Competitor pricing and demand elasticity directly influence demand calculations.

Challenges:

* Estimating demand elasticity accurately.
* Managing data inputs and ensuring algorithm stability over time:

Social Network Analysis

Task 1: Model the Social network.

Graph Model:

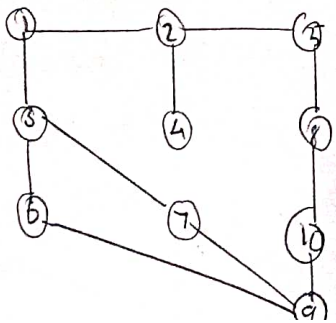
- Nodes: Represent users.
- Edges: Represent connection between user.

Task 2: Implement the PageRank Algorithm.

Pseudocode:

```
function Pagerank (graph, num_iteration, damping_factor):
    Initialize rank for each node to 1/ total_nodes.
    for each iteration. in num_iterations:
        new_rank = empty dictionary.
        for each node in graph:
            rank_sum = 0:
            for each incoming_node connected to node:
                rank_sum += rank [incoming_node] /
                    number_of_outgoing_edges [incoming_node]
        new_rank [node] = (1 - damping_factor) / total_nodes +
                          damping_factor * rank_sum:

        rank = new rank
    return rank.
```



Python implementation:

```
def pagerank (graph, num_iteration = 100, dumping_factor = 0.85):
    total_nodes = len (graph)
    rank = {nodes : 1/ total_nodes for node in graph}
for - in range (num_iteration):
    new_rank = {}
    for node in graph:
        rank_sum = sum [rank [incoming] / len [graph [incoming]]
                   for incoming in graph if node in graph [incom]]
    new_rank [node] = (1 - damping_factor) / total_nodes +
                      damping_factor * rank_sum
    rank = new_rank
    return rank.
graph = {
    'A' : {'B', 'C'},
    'B' : {'C'},
    'C' : {'A'},
    'D' : {'C'}
}

page_rank_result = pagerank (graph)
print ("Page rank :", page_rank_result).
```

Compare PageRank and Degree Centrality:

Degree Centrality:

* Count the number of direct connection as node has.

Comparison:

* Pagerank consider the equality of connection, giving more weight to connecting from influential nodes
* Degree centrality simply count connection, without considering their importance.

Example Degree Centrality Calculation:

```
def degree_centrality(graph):
    return {node: len(connection) for
        node, connection in graph.items()}
degree_centrality_result = degree_centrality
                           (graph)
Print("Degree Centrality:", degree_centrality
                           result)
```

Graph model: node represent user edges represent connections.

Pagerank Algorithm:
Implemented to identify influential users.

comparison:
Pagerank provide a more nuanced view of influence than degree centrality.

Reasoning:

* Pagerank is effective for identifying influential user because it considered both the quantity and quality of connection taking into account the influence of connected nodes
* Degree centrality measures direct connection, which may not always correlate with influence
* Usage Scenarios:
    * Pagerank is preferred in network where the influence of connection matter such as social media
* Degree centrality is useful for simpler analyses when only connection counts are relevant.

Fraud Detection in Finacial Transactions

Task1 : Design a greedy algorithm:

Pseudocode:

```
Function fraudDetection (transaction, rules):
    flagged_transaction = []
    for transaction in transaction:
        if transacting.amount > rules.max maunt:
            flagged_transaction.append(transaction)
        elif transaction.location_changes > rules.max_location_changes:
            flagged transaction.appnd (transaction)
        elif transaction.time_differece < rules.min time_difference:
            flagged_transaction.append(transaction)
    return flaged_transaction.
```

Implementation:

```
class Transaction:
    def __init__(self, amount location_changes, time_difference):
        self.amount = amount
        self.location_changes = location_changes
        self.time_difference = time_difference
    def fraud_detection (transaction, rules):
        flagged_transaction = []
        for transection in transaction:
            if transaction.amount > rules["max_amount"]:
                flagged_transaction.append(transection)
            elif transaction.location_changes > rules["max_location_changes"]:
                flagged_transaction.append(transaction)
            elif transaction.time_difference < rule["min_time_difference"]:
                flagged_transactions.append(transaction)
        return flagged_transaction.
    rules = {
            "max_amount": 10000,
            "max_location_change": 3,
            "min_time_difference": 10
        }
    transactions = [
            Transaction (15000, 1, 20),
            Transaction (5000, 4, 5),
            Transaction (2000, 2, 15)
        ]
    flagged = fraud_detection (transaction, rules
    print ("Flagged transaction :", flagged).
```

Performance Evaluation

* Precision: Proportion of correctly flagged fraudulent transaction:

$$\text{Precision} = \frac{TP}{TP+FP}$$

* Recall: Proportion of actual fraudulent transaction correctly flagged:

$$\text{Recall} = \frac{TP}{TP+FN}$$

* F1 Score: Harmonic mean of Precision and Recall.

$$\text{F1 score} = \frac{2 * \text{Precision} * \text{recall}}{\text{Precision} + \text{Recall}}$$

Suggestion and implementation of Improvement:

* Machine learning Integration: use historial data to train a model for more nuanced decision making

Anomaly Detection:

Implement algorithm like clustering or statical analysis to identify outliers.

Feature Engineering:

Incorporate addition transection features like users behaviour. Patterns

Reasoning:

* Greedy algorithm: suitable for real time detection as it processes transaction based on predefined rules, ensuring speed and simplicity.

* Speed vs Accuracy: Greedy algorithm prioritize speed, which is crucial for real time application. Accuracy may be enhanced through rule refinement or integrating machine learning model.

* Trade-offs:

Balancing speed and accuracy. involves using rules for quick decis while more complex methods can be applied in parallel for in-depth analysis.

Real-Time Traffic Management System:

Design a Backtracking Algorithm:

Pseudo code:

```
Function Optimize_Traffic_Light(intersection, max_green_time,
                                min_green_time):
    best_configuration = None
    best_flow = int

    function backtrack(intersection, current_configuration):
        if intersection == total_intersections:
            flow = simulate_traffic(current_configuration)
            if flow > best_flow:
                best_flow = flow
                best_configuration = current_configuration.copy()
            return
        for green_time in range(min_green_time, max_green_time+1):
            current_configuration[intersection] = green_time
            backtrack(intersection+1, current_configuration)
        current_configuration[intersection] = 0

    backtrack(0, [0] * total_intersection)
    return best_configuration
```

Implementation:

```
def simulate_traffic(configuration):
    return sum(configuration)
def optimize_traffic_lights(intersections, max_green_time,
                            min_green_time):
    best_configuration = None
    best_flow = float('-inf')
    total_intersection = len(intersection)
def backtrack(intersection, current_configuration):
    nonlocal best_configuration, best_flow
    if intersection == total_intersection:
        flow = simulate_traffic(current_configuration)
        if flow > best_flow:
            best_flow = flow
            best_configuration = current_configuration.copy()
        return
    for green_time in range(min_green_time, max_green_time+1):
        current_configuration[intersection] = green_time
        backtracking(intersection+1, current_configuration)
        current_configuration[intergection] = 0
backtrack(0, [0] * total_intersection)
return best_configuration
intersection = ['A', 'B', 'C']
max_green_time = 60
min_green_time = 30
best_config = optimize_traffic_light(intersection, max_green_time, min_green
```

Print("Best configuration - best_configuration)

Task 2: simulation and Performance Analysis:

Simulation:

* Use the optimized traffic light configuration in a traffic model.
* Measure traffic flow matrics such as average vechicle wait time and throughput.

Performance analysis:

```
def fixed_time_simulation():
    return 100
optimized_flow = simulate_traffic(best_config)
fixed_flow = fixed_time_simulation()
Print("Optimized traffic flow:", optimized_flow)
Print("Fixed_time traffic Flow:", fixed_flow)
```

Comparison with Fixed-time traffic light system:

* Evaluate difference in traffic metrics between optimized and fixed time system.
* Metrics include reduced congestion, improved flow, and decreased wait times: