Data Analysis & Algorithm for Backtracking - Assignment.

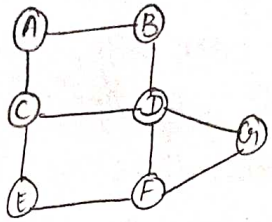
Sub-code: CSA0677.

Name: A. Dhivya
Reg. NO: 192311143
Subject: DAA.
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's Algorithm!

Pseudocode:

```
function Dijkstra (graph, start node):
Initialise distance with infinity, except start_node with 0
Initialize priority queue and add start_node with distance 0.
Initialise previous_nodes map to reconstruct paths
while priority_nodes map to reconstruct paths:
Remove current_node from the queue
for each neighbour of current_node:
calculate new_distance = distance [current node] + edge_weight
if new_distance < distance [neighbour]:
update distance [neighbour] to new_distance.
update previous_noder [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.heapop (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'
print ("Distance: ", distances)
Print ("Previous node: ", Previous_nodes).
```

Analysis of Algorithm efficiency and Potential improvement:

efficiency:

* Dijkstra's 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 travel times.

Potential Improvement:

* A Algorithm: Incorporates heuristics to improve performance in 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's 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 routes based on travel time.
* Assumptions include stable travel times and no negative weights. which fit most real-word Road network.
* variation in road condition, such a traffic congestion and closures, could impact travel times, requiring dynamic updates to the graph's weight for occurate route optimization.

Dynamic Pricing Algorithm for E-commerce:

Task 1: Design a dynamic program Algorithm:

function Dynamic Pricing (Product, periods, inventory, competitor_Price, demand_elasticity):

Initialise dp table with dimension [products][periods]

Initialize optimal_prices. 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:

maxprofit = revenue - cost

best_price = price.

dp[product][Period] = max_profit

optimal_prices[Product][period] = best_price

return dp, optimal_prices.

Python Implementation:

```
def calculate_demand(price, demand_elasticity,
                     competitor_price):
    base_demand = 100
    demand = base_demand(1 - demand_elasticity(
        Price - competitor_price) / competitor_
                                        prices)
    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_prices[i])
revenue = price * min(expected_demand_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.
```

```
Products ['A', 'B']
Periods = 3
inventory = [50, 40]
competitor_prices = [30, 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:

Prices 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.

Benefits:

* Maximize Revenue by adjusting price based on Realtime factors
* Increase Competitiveness by Reponding to market conditions.

Drawbacks:

* Complexity in implementation & maintenance.
* Potential customer dissatisfaction if prices fluctuate frequently.

Challenges:

* Estimating demand elasticity accurately.
* managing data inputs and ensuring algorithm stability overtime.

Reasoning:

Justification for dynamic Programming:

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

Incorporating factors:

* Inventory level are considered by limiting Revenue based on available stock.
* competitor pricing and demand elasticity influence demand calculations.

Social Network Algorithm:

Task 1: Model the Social Network:

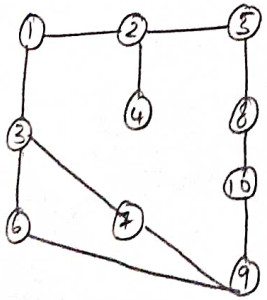
Graph model:

* Nodes: Represent users
* Edges: Represent connection b/w user:

Task 2: Implement the Page Rank Algorithm

Pseudocode:

```
function Page rank (graph, num_iteration, dumping factor):
Initialise rank for each node
  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[incoming])
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 = Page rank (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 considers 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
      nodes, 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 provides a more 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 prefered in network where the influence of connection matter such as social media.
* Degree centrally is useful for simpler analyses where only connection counts are relevant.

Fraud Detection in Financial Transactioning:

Task: 1: Design a greedy Algorithm.

Pseudocode:

```
Function Fraud detection (transaction, rules):
    flagged_ transaction = []
    for transaction in transaction:
    if transacting. amount > rules. max amount:
      flagged_ transaction. append (transaction)
    elif transaction. location_ changes > rules. max_ location_ changes:
    flagged transaction. append (transaction)
    elif transaction. time_difference < rules. min time_ difference:
      flagged_ transaction. append (transaction)
    return flagged_ 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 transaction in transaction.
    if transaction. amount > rules ["max_ amount"]:
      flagged_ transaction. append (transaction)
    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 transaction": 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 fraudlent transaction correctly flagged.

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

* F1 Score: Harmonic mean of Precision and Recall.

$$\text{F1 Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Suggestion and implementation of Improvement:

* Machine Learning Integration: Use Historical 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 transaction features like user behaviour pattern.

Reasoning:

* Greedy Algorithm: Suitable for real time detection as it proces 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 decision while more complex methods can be applied in parallel for in-depth Analysis.

Real-Time Traffic Management System:

Design a Backtracking Algorithm:

Pseudocode:

```
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)
        best_flow = flow
        best configuration = current_configuration.copy()
    return.
  for green_time in range (min_green_time, max_green_time+1),
    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_gree_time):
  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()
  current_configuration [intersection] = green_time.
  backtracking (intersection + 1, current_configuration)
backtrack (0, [0] ^ total_intersection)
return best_configuration.
  max_green_time = 60
  min_green_time = 30
  Print ("But configuration_ best_configuration).
```

Task 2 : Simulation and Performance Analysis:

Simulation:

* Use the optimized traffic light configuration in a traffic model.
* Measure traffic flow matrices such as average vehicle 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 diff in traffic metrices btw optimized and fixed time system.
* Metrices include congestion, improved flow, and decreased wait times.