192365009
SUPRAJA. V.S.

CSA0676

Design and Analysis of Algorithms

Problem 1: Optimizing Delivery Routes [case study]

Scenario: you are working for a logistics company that wants to optimize its delivery routes to minimize fuel consumption and delivery time. The company operates in a city with a complex road network.

Tasks:

* Model the city's road network as a graph where intersection are roads are edges with weight representing travel time.
* Implement Dijkstra's algorithm to find the shortest path
* Analyse the efficiency of algorithm and discuss any potential improvements.

Solution:

* Why Dijkstra's algorithm is suitable: It efficiently finds the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights, which is typically the case for travel time.
* Assumption made: the weights are non-negative. Road conditions such as traffic and road closure would affect travel time.

1. Graph model of the city's Road Network.

we will represent the city's road network as a graph $G = (V, E)$:

* V is a nodes representing intersections.
* E is a set of edges representing roads.

2. Pseudocode and Implementation of Dijkstra's algorithm.

Pseudocode:

```
function dijkstra(graph, source):
    dist[source] := 0
    for each vertex v in graph:
        if v ≠ source:
            dist[v] := infinity
    add v to priority queue Q
    while Q is not empty:
        u := vertex in Q with min dist[u]
    remove u from Q
    for each neighbour v of u:
        alt := dist[u] + length(u,v)
        if alt < dist[v]:
            dist[v] := alt
            prev[v] := u
    return dist, prev.
```

Implementation:

```
import heapq
def dijkstra(graph, start):
    priority_queue = []
    distance = {vertex: float('infinity')
    for vertex in graph}
    distances[start] = 0
    heapq.heappush(priority_queue, (0, start))
```

```
while priority_que:
    current_distance, current_vertex = heapq.heappop.
if current_disbance > distances[current_vertex]:
        continue
for neighbor, weight in graph[current_vertex].items():
        distance = current_distance + weight.

return distances.
```

Analysis of the algorithms efficiency & potential improvements.

Time complexity : using a priority queue is $O((V+E)\log V)$.

Space complexity : using a priority que is $O(V+E)$

potential improvements:

1. A Algorithm: for more efficient pathfinding with a heuristic the A* algorithm can be used.
2. Dynamic Update: If travel time change freq. due to traffic a dynamic algorithm can be used.
3. Bidirectional Dijkstras: Running this algorithm from the source and target nodes can sometimes reduce computation time

Conclusion:

Dijkstra's algorithm is a robust and efficient method for finding the shortest paths in a weighted graph with non-negative weights. It suits the problem of optimizing delivery routes in a city's road network.

Problem 2: Dynamic Pricing algorithm for E-commerce.

Scenario: An e-commerce company wants to implement a dynamic pricing algorithm to adjust a prices of products in real-time based on demand.

Tasks:

1. Design a dynamic programming algorithm.
2. Consider factors such as inventory levels, competitor prices and demand elasticity.
3. Test your algorithm with simulated data and compare its performance with a simple static price strategy.

Solution:

1. Dynamic programming is suitable for this problem as it helps breaking down the complex problem of pricing optimization.
2. The algorithm consider inventory levels, competitor prices, demand elasticity to adjust prices dynamically.
3. Challenges: include accurately modelling demand elasticity, handling the computational complexity.

Pseudo code:

```
function dp(products, periods, dE, cp, inventory):
    dp = array of dimensions(products, periods)
    for t in range(periods):
        for p in products:
            mp = 0
            bp = 0
```

```
        for price in range(mp, mp, ps):
            ed = dE(p, price, CP[p][t])
            if ed = Inventory[p][t]:
                profit = price * expected demand
            if t < periods - 1: profit += dp[p][t+1]
            if profit > mp:
            mp = profit
            bp = price.
        return optimal_prices.
```

Implementation in Python:

```
import numpy as np
class DP:
  def __init__(self, mp, map, ps):
      slef.mp = mp
      slef.map = map
      slef.ps = ps.
  def de(self, product, price, competion):
      bd = 100
      ps = 0.5
      cs = 0.3
      return max(bd - ps * price + cs * cp, 0)
def dp(self, product, periods, cp, inventory):
      dp = np.zeros(len(products), periods')
      bp = np.zeros(len(products), periods')
      for t in range(periods - 1, -1, -1)
      for i, product in enumerate:
            mp = 0
            bp = 0
```

if profit > max profit:

Mp = profit

pp = price

return optimal prices.

Simulation Results:

* Static pricing Revenue : $xx (based on the static)
* Dynamic pricing Revenue : $yy (based on dynamic pricing)

Benefits:

1. Dynamic pricing can lead to higher revenue.
2. By considering competitor prices, the algorithm ensures competitiveness in the market.
3. Dynamic pricing helps in managing inventory levels.

Drawbacks:

1. Dynamic pricing algorithms can be complex.
2. Frequent price changes might lead to customer dissatisfaction.
3. The accuracy of the algorithm heavily depends on the quality.

Conclusions:

Dynamics pricing algorithm offer significant benefits in terms of increased revenue and better inventory management compared to static pricing strategies, However, they come with challenges related to complexity.

Problem 3: Social Network Analysis (case study)

Scenario: A social media company wants to identify influential users within its network to target for marketing campaigns.

Tasks:

1. model the social network as a graph.
2. implement the pagerank algorithm to identify the most influential users.
3. compare the results of Page rank with a simple degree.

Solution:

1. Page Rank is effective for identifying influential users as it considers the quality of connections.
2. Difference between page rank and Degree centrality. Degree centrality simply counts the number of connections a user has.

Pseudocode:

```
function PR(graph, alpha, MI, tol):
    N = number of nodes in graph
    rank = dict to set 1/N
    for i in range(MI):
        new_rank = dict with each node set 1/N
    for node in graph:
        for neighbor in graph[node]:
            rank[neighbor] += alpha * rank[node].
    if sum(abs(new_rank[node] - rank[node]) for node in graph)
    < tol:
        break
    rank = new_rank
    return rank.
```

Implementation in python:

```
import numpy as np
def pr( graph, alpha = 0.85, MI = 100, tol = 1e-6):
    nodes = list (graph.nodes)
    N = len (nodes)
    rank = {node : 1 /N for node in nodes}
    for _ in range (MI):
        nr = {node : (1-alpha)/N
    for node in nodes:
        for neighbor in graph. neighbors (node)
        new_rank [neighbor] += alpha * rank [node]
        break
    rank = new_rank
return rank.
```

Degree Centrality calculation:

```
degree_centrality = nx.degree_centrality (G)
print (degree_centrality)
```

s:

Considers both the quantity and quality of connections. A node connected to other highly connected nodes will have a higher PageRank score.

2. Counts the number of connections directly. A node with more direct connections will have a higher degree.

Conclusion:

Page ranks provides a more nuanced measure of influence by considering the quality of connections, making it more effective in identifying influential users in a social network. Degree centrality is simpler and easier.

Problem 4: Fraud Detection in Financial ~~accounting~~ transactions:

Scenario:

A financial institution wants to develop an algorithm to detect fraudulent transaction in real time.

Tasks:

1. Design a greedy algorithm to flag potentially fraudlent transaction based on set of predefined rules.
2. Evalute the algorithm's performance using historical transaction data and calculate metrics such as precision, recall.
3. Suggest and implement potential improvements to the algorithm.

Solution:

Task 1:

predefined rules:

* unusually large transactions (Total Limit = 1000)
* transactions from multiple location in short time

In adressing the problem of fraud detection in financial transaction I have devised a greedy algorithm based on predefined rules.

Task 2:

Five transactions are considered as the input data for the program has predefined whether the transaction is fraudlent or not. The data contains the amount and location of the transactions. Parameters such as precision, recall and F1 score are calculated using true positive and false negative and true negative, true positive

Transaction 1: $500 Location: A

Transaction 2: $2000 Location: B

Tsanaction 3: $500 Location: A

Transaction 4: $1200 Location: C

Transaction 5: $700 Location: B

* precision: $\frac{TP}{TP+FP} = \frac{2}{2+1} = \frac{2}{3} \approx 0.67 \approx 1.0$

* Recall: $\frac{TP}{TP+FN} = \frac{2}{2+0} = 1.0$

* F1 Score = $\frac{2 \times \text{precision} \times \text{recall}}{\text{Precision} + \text{Recall}}$

$= \frac{2 \times 0.67 \times 1.0}{0.67 + 1}$

$= \frac{1.34}{1.67}$

$= 0.80 \approx 1.0$

Implementation:

```
Class fd:
    def __init__(self, ma, loca):
        self.maxamount = ma
        self.loc = loc

def fraud(self, trans):
    if trans['amount'] > self.maxamount:
        return true.
    history = transaction['recent_transaction']
    loc = set(t['loc'] for t in history)

def evaluate(self, history):
    tp = 0
    fp = 0
    fn = 0
    tn = 0

for transaction in history:
    pred = self.fraud(trans)
    actl = trans['fraud']

    if prediction and actual:
        tp += 1
    elif prediction and not actual:
        fp += 1
    elif not prediction and not ac:
        tn += 1

precision = tp/(tp+fp) if (tp+fp) > 0 else 0
recall = tp/(tp+fn) if (tp+fn) > 0 else 0
f1 = 2 * (precision * recall)/(precision + recall) if
(precision + recall) > 0 else 0

return precision, recall, f1
```

Pseudo code:

```
for each transaction in sorted history:
    SET prediction = self.fraud(transaction)
    SET actual = transaction['fraud']
    IF prediction AND actual:
        ~~if pre~~ increment tp
    elif prediction AND NOT actual:
        INCREMENT fp
    Elif prediction AND NOT actual:
        INCREMent in

SET precision = tp/(tp+fp) IF (tp+fp) > 0 else 0
SET recall = tp/(tp+fn) IF (tp+fn) > 0 else 0
SET f1 = 2 * (precision * recall)/(precision + recall) IF
(precision + recall) > 0 else 0

Return precision, recall, f1
```

improvements:

1. Tune the threshold value
2. consider using machine learning algorithms like decision trees, random forest.
3. Add more features to the transaction data, such as user behaviour, ip addrs and device information.
4. Implement anomaly detection techniques to identify.

Alternative algorithms:

1. Multi-stage Greedy algorithm.
2. weighted Greedy algorithm.
3. Greedy Algorithm with Historical Comparision
4. Context-Aware Greedy Algorithm.

Problem 5: Real-time traffic Management system.

Scenario:

A city's traffic management department wants to develop a system to manage traffic lights in real-time to reduce congestion.

Tasks:

1. Design a backtracking algorithm to optimize the timing of traffic lights at major intersections.
2. Simulate the algorithm on a model of the city's traffic network and measure its impact on traffic flow
3. Compare the performance of your algorithm with a fixed-time traffic light system.

Solution:

1. Backtracking is suitable for this problem because it allows for exploring all possible configurations of traffic light timing systematically. It can handle complex constraints.
2. Real-time traffic management involves handling dynamic and unpredictable traffic patterns. It requires balancing various factors such as traffic density, intersection wait times and overall traffic

Task 1:

Pseudocode:

```
function optimize_traffic_lights(intersections, max_time, min_
    best_config = None
    best_flow = infinity
function backtrack(current_index, current_config):
    if all intersections are configured:
        current_flow = simulate_traffic.
```

```
if current_flow < best_flow:
    best_flow = current_flow
    best_config = current_config
    return
for green_time in range(min_green, max_green):
current_config[current_intersection] = green_time
    next_intersection = get_next_intersection()
return best_configuration
```

Task 2: Implementation in Python

```
def optimize_traffic(intersections, max_green, min_green):
    best_config = None
    best_flow = float('inf')
    def backtrack(current, current_config):
        nonlocal best_config, best_flow
        if current == len(intersections):
            current_flow = simulate_traffic(current_config)
            if current_flow < best_flow:
                best_flow = current_flow
                best_config = current_config.copy()
            return
        for green in range(min_green, max_green+1):
            return best_configuration
```

```
import random
return random.uniform(0, 100)
```

Task 3: Simulation & analysis

Simulation:

To simulate the algorithm, we need to create a model of the city's traffic network. For simplicity, lets assume a small network with few intersection and simulate the impact of different traffic light configuration.

Analysis:

1. The optimized traffic light configuration significantly reduces total delay compared to the fixed-time configuration by adapting to real-time traffic conditions.
2. While simpler to implement, the fixed-time configuration does not account for dynamic traffic patterns, leading to suboptimal performance in terms of reducing congestion.