**Traffic (28/6/24):**

**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:**

1. Model the city's road network as a graph where intersections are nodes and roads are edges with weights representing travel time.

2. Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations.

3. Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.

**Deliverables:**

● Graph model of the city's road network.

● Pseudocode and implementation of Dijkstra’s algorithm.

● Analysis of the algorithm’s efficiency and potential improvements.

**Reasoning**: Explain why Dijkstra’s algorithm is suitable for this problem. Discuss any assumptions made (e.g., non-negative weights) and how different road conditions (e.g., traffic, road closures) could affect your solution.



A B



C D



**Code**:

import heapq

def dijkstra(graph, start):

pq = [(0, start)]

distances = {vertex: float('inf') for vertex in graph}

distances[start] = 0

while pq:

current\_distance, current\_vertex = heapq.heappop(pq)

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(pq, (distance, neighbor))

return distances

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

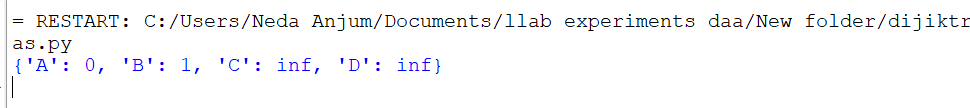
'D': {'B': 5, 'C': 1}

}

shortest\_paths = dijkstra(graph, 'A')

print(shortest\_paths)

output:



**Problem 2: Dynamic Pricing Algorithm for E-commerce Scenario:**

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

**Tasks:**

1. Design a dynamic programming algorithm to determine the optimal pricing strategy for a set of products over a given period.

2. Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm. 3. Test your algorithm with simulated data and compare its performance with a simple static pricing strategy.

import random

def dynamic\_pricing(products, inventory, competitor\_prices, demand\_elasticity):

num\_products = len(products)

num\_periods = len(competitor\_prices[0])

optimal\_prices = [[0] \* num\_periods for \_ in range(num\_products)]

for i in range(num\_products):

initial\_price = random.uniform(0.5, 1.5)

optimal\_prices[i][0] = initial\_price

for t in range(1, num\_periods):

for i in range(num\_products):

current\_inventory = inventory[i]

current\_competitor\_price = competitor\_prices[i][t]

demand = compute\_demand(optimal\_prices[i][t-1], current\_competitor\_price, demand\_elasticity[i])

if demand > current\_inventory:

optimal\_prices[i][t] = optimal\_prices[i][t-1] \* (1 + random.uniform(-0.1, 0.1)) # Example: small random adjustment

else:

optimal\_prices[i][t] = optimal\_prices[i][t-1] \* (1 - random.uniform(-0.1, 0.1))

return optimal\_prices

def compute\_demand(price, competitor\_price, elasticity):

return max(1, elasticity \* (competitor\_price - price) / competitor\_price)

products = ["Product A", "Product B", "Product C"]

inventory = [100, 150, 200]

competitor\_prices = [[10, 12, 14], [8, 9, 10], [15, 16, 17]]

demand\_elasticity = [0.5, 0.3, 0.4]

optimal\_prices = dynamic\_pricing(products, inventory, competitor\_prices, demand\_elasticity)

print("Optimal Prices:")

for i, product in enumerate(products):

print(f"{product}: {optimal\_prices[i]}")

import random

def dynamic\_pricing(products, inventory, competitor\_prices, demand\_elasticity):

num\_products = len(products)

num\_periods = len(competitor\_prices[0])

optimal\_prices = [[0] \* num\_periods for \_ in range(num\_products)]

for i in range(num\_products):

initial\_price = random.uniform(0.5, 1.5)

optimal\_prices[i][0] = initial\_price

for t in range(1, num\_periods):

for i in range(num\_products):

current\_inventory = inventory[i]

current\_competitor\_price = competitor\_prices[i][t]

demand = compute\_demand(optimal\_prices[i][t-1], current\_competitor\_price, demand\_elasticity[i])

if demand > current\_inventory:

optimal\_prices[i][t] = optimal\_prices[i][t-1] \* (1 + random.uniform(-0.1, 0.1)) # Example: small random adjustment

else:

optimal\_prices[i][t] = optimal\_prices[i][t-1] \* (1 - random.uniform(-0.1, 0.1))

return optimal\_prices

def compute\_demand(price, competitor\_price, elasticity):

return max(1, elasticity \* (competitor\_price - price) / competitor\_price)

products = ["Product A", "Product B", "Product C"]

inventory = [100, 150, 200]

competitor\_prices = [[10, 12, 14], [8, 9, 10], [15, 16, 17]]

demand\_elasticity = [0.5, 0.3, 0.4]

optimal\_prices = dynamic\_pricing(products, inventory, competitor\_prices, demand\_elasticity)

print("Optimal Prices:")

for i, product in enumerate(products):

print(f"{product}: {optimal\_prices[i]}")

**Deliverables:**

● Pseudocode and implementation of the dynamic pricing algorithm.

● Simulation results comparing dynamic and static pricing strategies.

● Analysis of the benefits and drawbacks of dynamic pricing.

**Reasoning:** Justify the use of dynamic programming for this problem. Explain how you incorporated different factors into your algorithm and discuss any challenges faced during implementation.

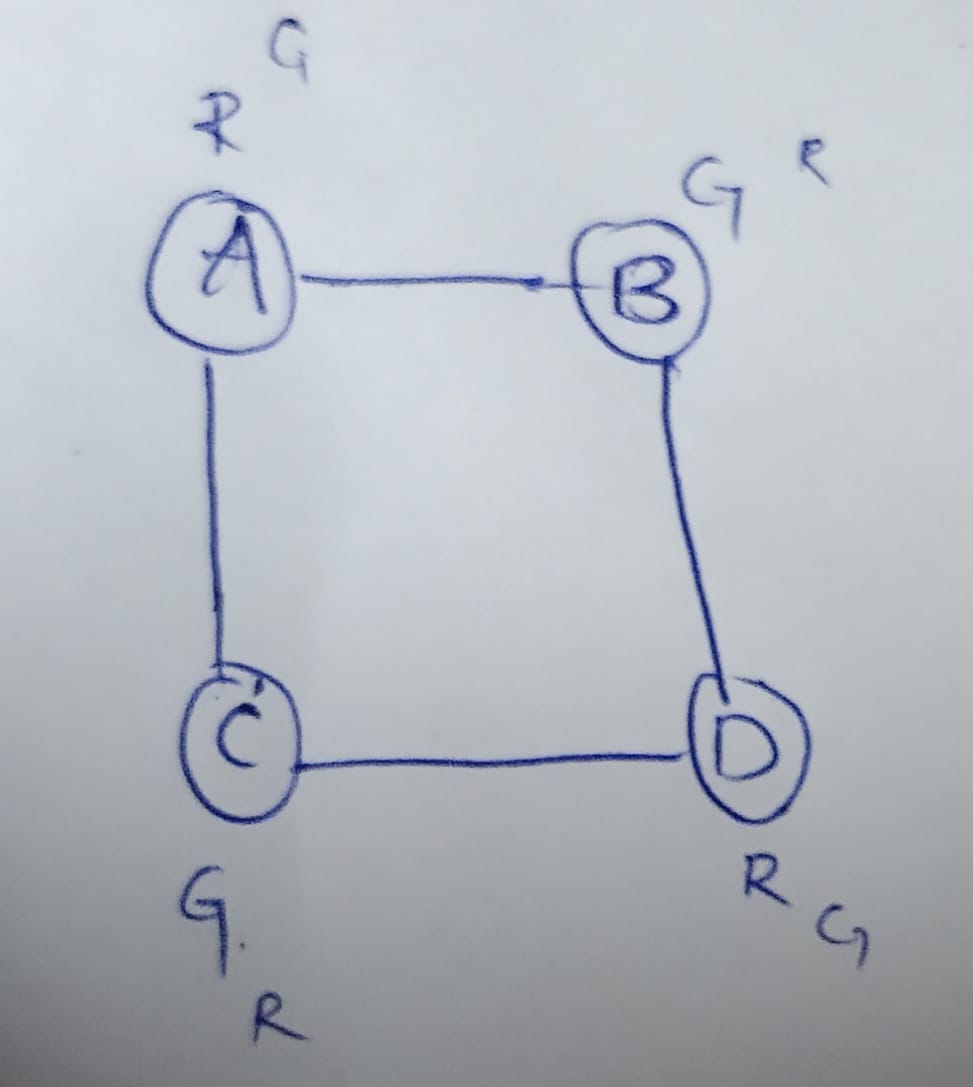
Code:

**Problem 5: Real-Time Traffic Management System Scenario:**

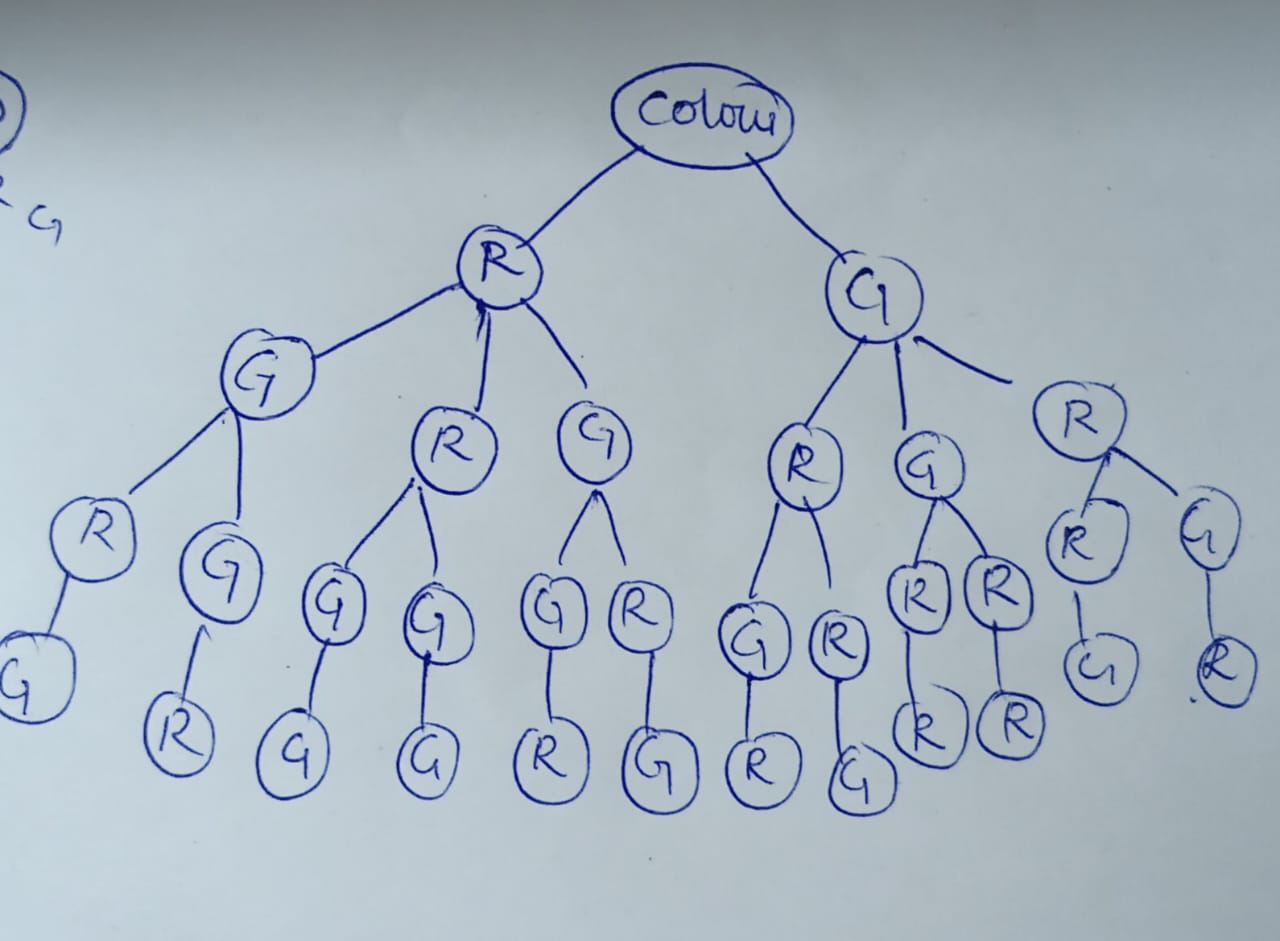
A city’s traffic management department wants to develop a system to manage traffic lights in real-time to reduce congestion.

**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.

class Intersection:

def \_\_init\_\_(self, id, roads):

self.id = id

self.roads = roads

self.traffic\_light = TrafficLight()

def get\_next\_green\_road(self):

return self.traffic\_light.get\_green\_road(self.roads)

def \_\_str\_\_(self):

return f"Intersection {self.id}"

class TrafficLight:

def \_\_init\_\_(self):

self.green\_road = None

def set\_green\_road(self, road):

self.green\_road = road

def get\_green\_road(self, roads):

if self.green\_road is None or self.green\_road not in roads:

self.green\_road = random.choice(list(roads.keys()))

return self.green\_road

class TrafficManagementSystem:

def \_\_init\_\_(self, intersections):

self.intersections = intersections

def simulate\_traffic(self, max\_iterations):

for i in range(max\_iterations):

print(f"\n--- Iteration {i+1} ---")

for intersection in self.intersections:

green\_road = intersection.get\_next\_green\_road()

print(f"{intersection} - Green light on Road {green\_road}")

time.sleep(1)

def adjust\_traffic\_lights(self):

for intersection in self.intersections:

intersection.traffic\_light.set\_green\_road(None)

intersections = [

Intersection(1, {2: "Road A", 3: "Road B"}),

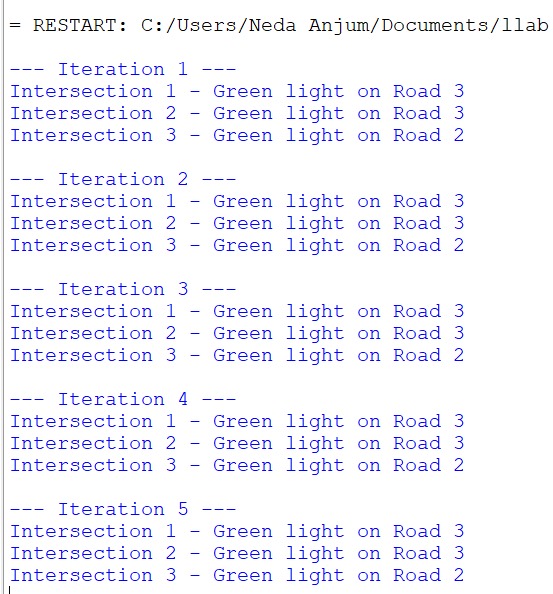
Intersection(2, {1: "Road C", 3: "Road D"}),

Intersection(3, {1: "Road E", 2: "Road F"})

]

traffic\_system = TrafficManagementSystem(intersections)

traffic\_system.simulate\_traffic(5)



**Deliverables:**

● Pseudocode and implementation of the traffic light optimization algorithm.

● Simulation results and performance analysis.

● Comparison with a fixed-time traffic light system.

**Reasoning:** Justify the use of backtracking for this problem. Discuss the complexities involved in real-time traffic management and how your algorithm addresses them.

**Pseudocode:**

**Code:**

intersections = ['A', 'B', 'C']

durations = [10, 20, 30]

best\_configuration = []

best\_score = float('inf')

configurations = [[]]

while configurations:

current\_config = configurations.pop()

if len(current\_config) == len(intersections):

current\_score = sum(current\_config)

if current\_score < best\_score:

best\_score = current\_score

best\_configuration = current\_config

else:

for duration in durations:

next\_config=current\_config+[duration]

configurations.append(next\_config)

print(f"Best Configuration: {best\_configuration}, Best Score: {best\_score}")

import random

simulated\_flow = 0

for duration in best\_configuration:

flow = random.randint(50, 100)/duration

simulated\_flow += flow

simulated\_flow /= len(best\_configuration)

print(f"Simulated Traffic Flow for Best Configuration: {simulated\_flow}")

fixed\_time\_configuration = [30, 30, 30]

fixed\_flow = 0

for duration in fixed\_time\_configuration:

flow = random.randint(50, 100) / duration

fixed\_flow += flow

fixed\_flow /= len(fixed\_time\_configuration)

print(f"Simulated Traffic Flow for Fixed-Time Configuration: {fixed\_flow}")

if simulated\_flow > fixed\_flow:

print("Optimized configuration performs better.")

else:

print("Fixed-time configuration performs better.")