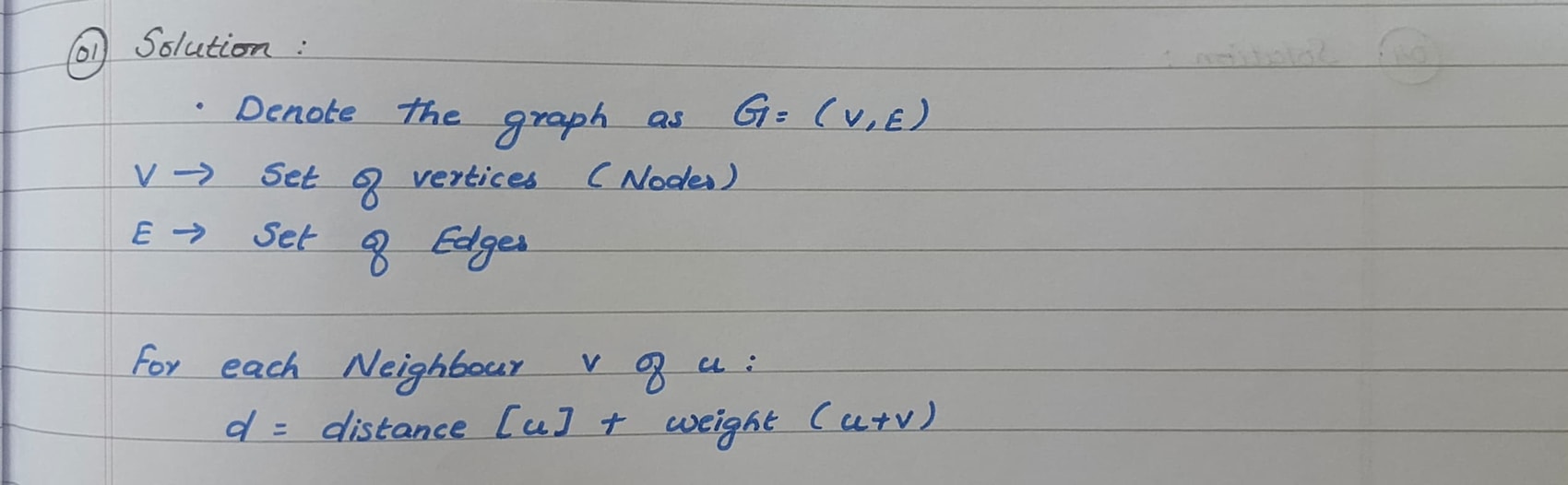
CSA 0674 - DAA ASSIGNMENT

## **Problem 1: Optimizing Delivery Routes (Case Study)**

### **Task 1: Model the City's Road Network as a Graph**

In this task, the city's road network is represented as a graph. Each intersection is a node, and each road connecting the intersections is an edge. The weight of each edge represents the travel time between the intersections, which can be influenced by factors such as traffic, road conditions, and speed limits.

A graph GGG is defined by a set of vertices VVV (intersections) and edges EEE (roads). Each edge eee has a weight w(e)w(e)w(e) which signifies the travel time. Formally, G=(V,E,w)G = (V, E, w)G=(V,E,w).



### **Task 2: Implement Dijkstra’s Algorithm**

Dijkstra’s algorithm is a well-known algorithm used for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. The algorithm works on graphs with non-negative weights and finds the shortest path from a single source node to all other nodes.

**Pseudocode for Dijkstra's Algorithm:**

function Dijkstra(Graph, source):

dist[source] := 0

for each vertex v in Graph:

if v ≠ source:

dist[v] := infinity

add v to Q

while Q is not empty:

u := vertex in Q with min dist[u]

remove u from Q

for each neighbor 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):

pq = []

heapq.heappush(pq, (0, start))

distances = {vertex: float('infinity') 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

### **Task 3: Analyze the Efficiency of the Algorithm**

**Efficiency Analysis:**

* Time Complexity: The time complexity of Dijkstra’s algorithm is O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges. This is due to the priority queue operations, which involve insertion and extraction in logarithmic time.
* Space Complexity: The space complexity is O(V)O(V)O(V) because it requires storage for the distance table and priority queue.

**Potential Improvements:**

* *A Algorithm:*\* This is an improvement over Dijkstra’s algorithm by using heuristics to estimate the shortest path, thereby potentially reducing the number of nodes processed.
* Dynamic Road Conditions: Incorporate real-time traffic data to adjust the weights of the edges dynamically, ensuring the shortest path reflects current conditions.

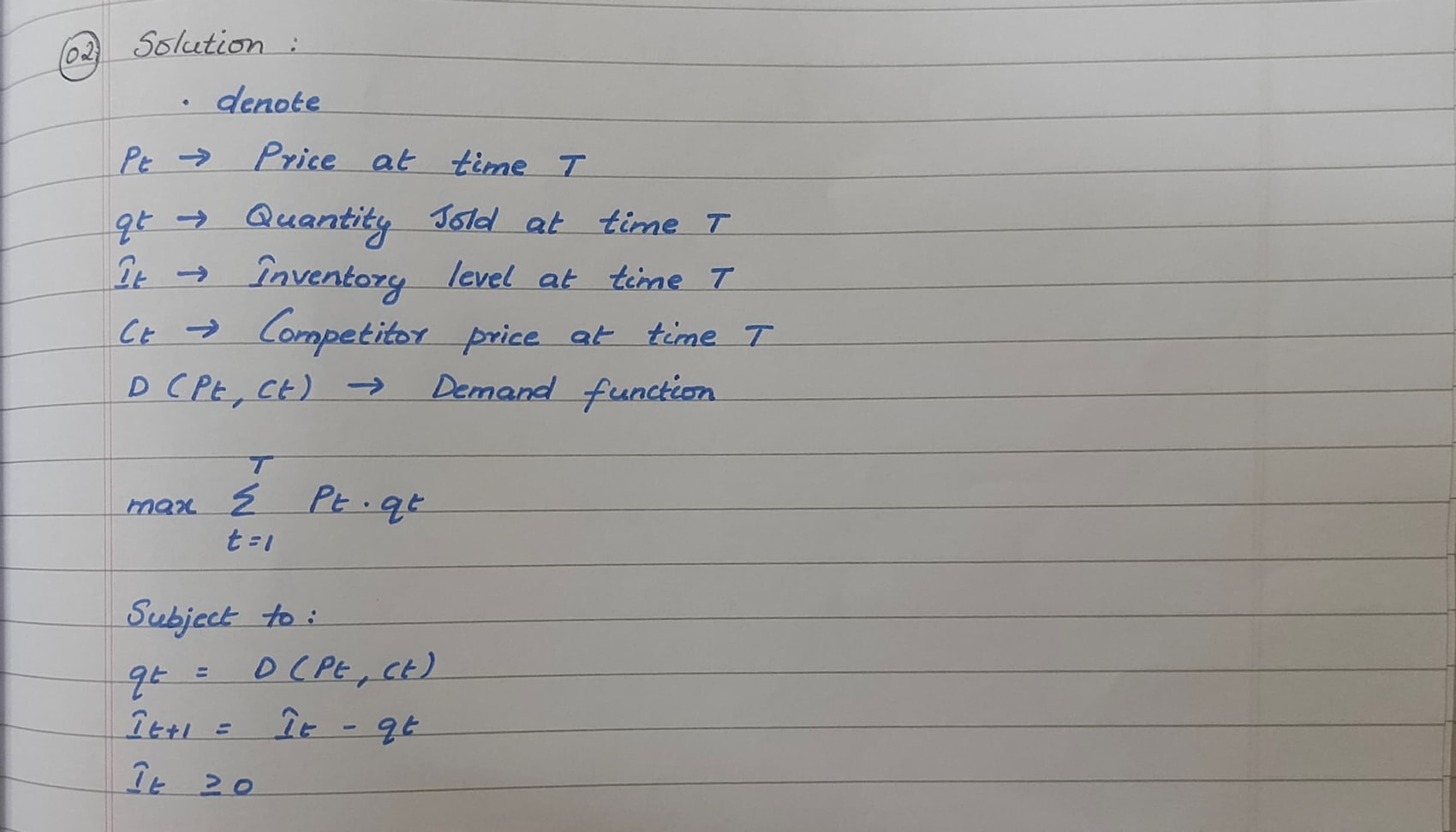
**Reasoning:**

Dijkstra’s algorithm is chosen for its simplicity and effectiveness in finding shortest paths in graphs with non-negative weights. However, it is static and does not account for real-time changes in traffic, which is a limitation that can be addressed by integrating dynamic data.

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

### **Task 1: Design a Dynamic Pricing Algorithm**

Dynamic pricing in e-commerce involves adjusting the prices of products based on various factors like inventory levels, competitor prices, and demand elasticity. This approach helps in maximizing revenue and staying competitive.



**Pseudocode for Dynamic Pricing Algorithm:**

function DynamicPricing(products, period, inventory, competitor\_prices, demand\_elasticity):

for t in period:

for each product in products:

optimal\_price := calculate\_optimal\_price(inventory[product], competitor\_prices[product], demand\_elasticity[product])

set price of product to optimal\_price

return prices

**Implementation:**

def calculate\_optimal\_price(inventory, competitor\_price, demand\_elasticity):

# Placeholder function for optimal price calculation

return competitor\_price \* demand\_elasticity

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

prices = {}

for t in period:

for product in products:

optimal\_price = calculate\_optimal\_price(inventory[product], competitor\_prices[product], demand\_elasticity[product])

prices[product] = optimal\_price

return prices

### **Task 2: Consider Factors in the Algorithm**

Incorporating factors such as inventory levels, competitor pricing, and demand elasticity is crucial for an effective dynamic pricing algorithm. These factors ensure that the prices set are competitive and responsive to market changes.

* Inventory Levels: Higher inventory might lead to lower prices to clear stock, while lower inventory could increase prices.
* Competitor Pricing: Prices need to be competitive to attract customers.
* Demand Elasticity: Reflects how sensitive the quantity demanded is to changes in price. Products with high elasticity might see significant changes in demand with small price adjustments.

### **Task 3: Test the Algorithm**

**Simulation Results:**

Simulation of the algorithm can be conducted by creating a test environment with historical sales data, competitor prices, and inventory levels. Comparing the performance of dynamic pricing against static pricing involves measuring metrics like revenue, sales volume, and market share.

**Benefits and Drawbacks:**

* Dynamic Pricing:
  + Benefits: Maximizes revenue, adapts to market conditions, and remains competitive.
  + Drawbacks: Requires complex calculations and real-time data, which can be resource-intensive.
* Static Pricing:
  + Benefits: Simple to implement and maintain.
  + Drawbacks: May not optimize revenue or adapt to market changes effectively.

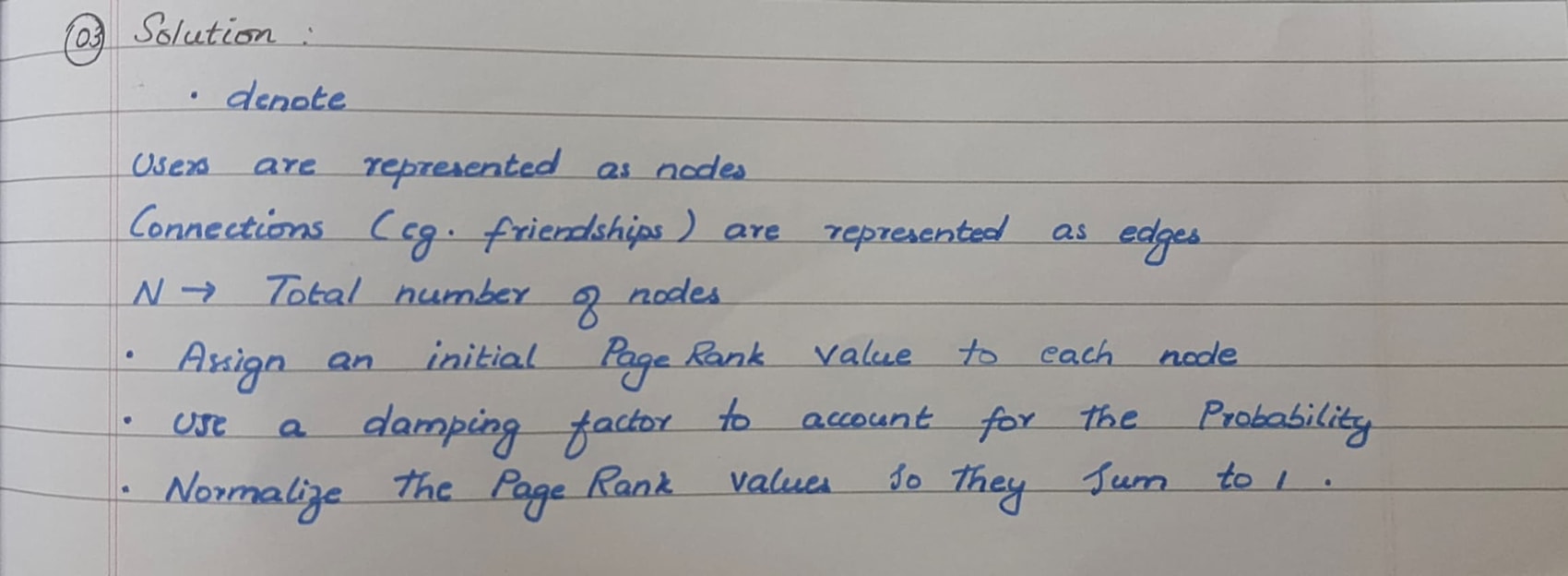
**Reasoning:**

Dynamic pricing is more effective in maximizing revenue and adapting to market conditions. While it is more complex, the benefits outweigh the drawbacks in competitive markets.

## **Problem 3: Social Network Analysis (Case Study)**

### **Task 1: Model the Social Network as a Graph**

In social network analysis, the network is modeled as a graph where users are represented as nodes and connections (e.g., friendships, followers) are represented as edges. This graph structure allows for the analysis of relationships and influences within the network.



### **Task 2: Implement the PageRank Algorithm**

PageRank is an algorithm initially used by Google to rank web pages. It can be applied to social networks to rank users based on their influence within the network.

**Pseudocode for PageRank Algorithm:**

function PageRank(Graph, d, iterations):

N := number of nodes in Graph

rank := [1/N] \* N

for i in 1 to iterations:

new\_rank := [0] \* N

for each node v in Graph:

for each neighbor u of v:

new\_rank[u] += d \* rank[v] / out\_degree[v]

new\_rank[v] += (1 - d) / N

rank := new\_rank

return rank

**Implementation:**

def pagerank(graph, d=0.85, iterations=100):

N = len(graph)

rank = {node: 1/N for node in graph}

for \_ in range(iterations):

new\_rank = {node: (1 - d) / N for node in graph}

for node in graph:

for neighbor in graph[node]:

new\_rank[neighbor] += d \* rank[node] / len(graph[node])

rank = new\_rank

return rank

### **Task 3: Compare PageRank with Degree Centrality**

**Comparison with Degree Centrality:**

* PageRank: Considers not just the number of connections a node has, but also the quality of those connections. A node connected to highly connected nodes will have a higher rank.
* Degree Centrality: Simply counts the number of connections a node has. It is a simpler measure but does not capture the influence of connections.

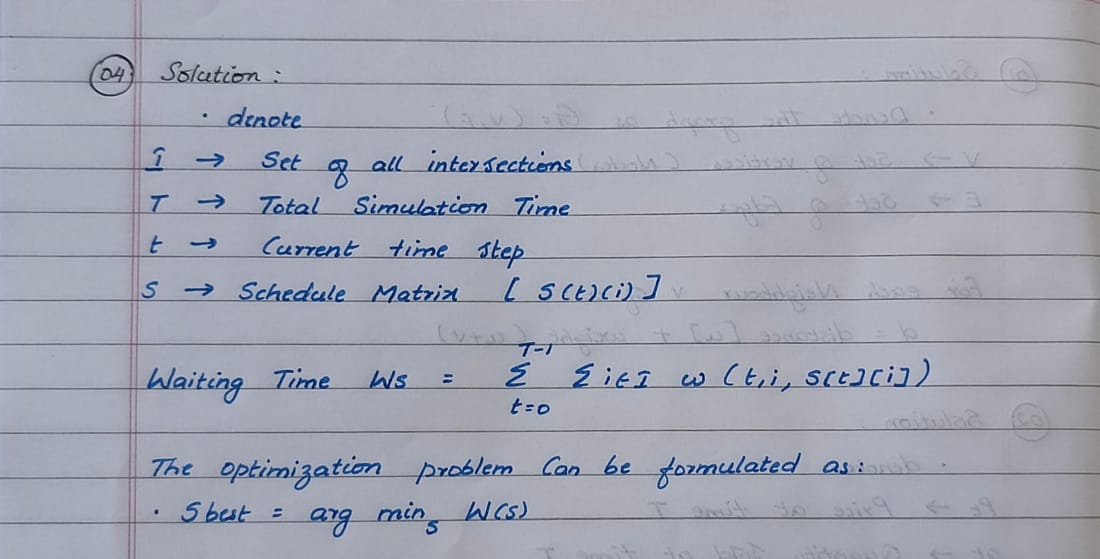
**Reasoning:**

PageRank is chosen for its ability to measure influence within the network, taking into account the importance of connections. Degree centrality, while simpler, is less informative in complex networks where the quality of connections matters.

## **Problem 4: Fraud Detection in Financial Transactions**

### **Task 1: Design a Greedy Algorithm for Fraud Detection**

Fraud detection algorithms aim to identify suspicious transactions based on predefined rules. A greedy algorithm can quickly flag transactions that meet any of these rules.



**Pseudocode for Fraud Detection Algorithm:**

function FraudDetection(transactions, rules):

flagged\_transactions := []

for each transaction in transactions:

if transaction meets any rule in rules:

add transaction to flagged\_transactions

return flagged\_transactions

**Implementation:**

def fraud\_detection(transactions, rules):

flagged\_transactions = []

for transaction in transactions:

for rule in rules:

if rule(transaction):

flagged\_transactions.append(transaction)

break

return flagged\_transactions

### **Task 2: Evaluate Algorithm Performance**

**Performance Evaluation:**

Evaluating the performance of the fraud detection algorithm involves calculating precision, recall, and the F1 score using historical transaction data.

* Precision: The proportion of flagged transactions that are actually fraudulent.
* Recall: The proportion of actual fraudulent transactions that are correctly flagged.
* F1 Score: The harmonic mean of precision and recall, providing a single measure of accuracy.

### **Task 3: Suggest Improvements**

**Potential Improvements:**

* Machine Learning Models: Train models using features from historical data to improve accuracy.
* Anomaly Detection Techniques: Use statistical methods to detect outliers that may indicate fraud.

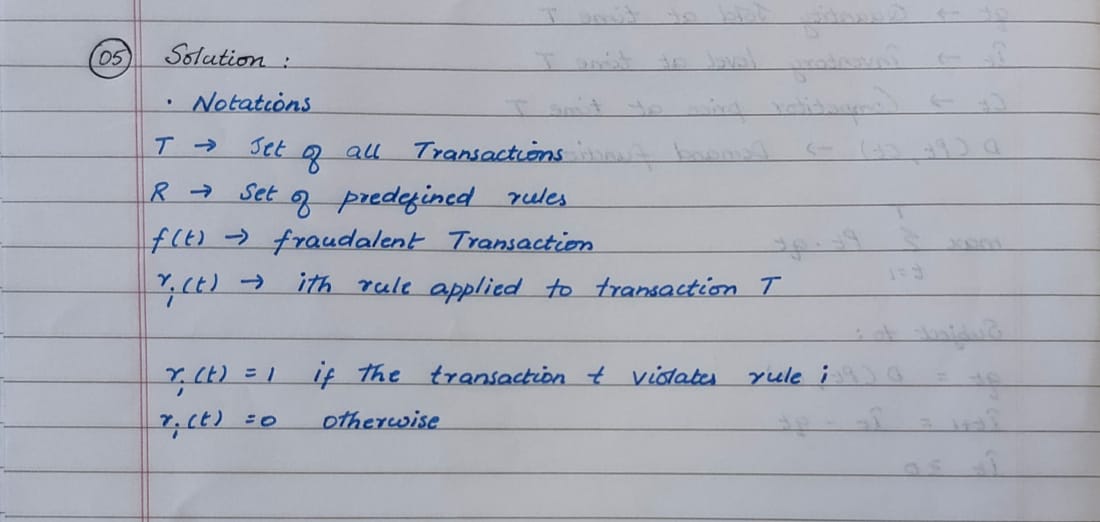
**Reasoning:**

The greedy algorithm provides a quick and simple method for fraud detection. However, integrating machine learning models and anomaly detection techniques can enhance the accuracy and adaptability of the system.

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

### **Task 1: Design a Backtracking Algorithm**

A backtracking algorithm can be used to optimize traffic light timings in real-time by exploring all possible configurations and choosing the one that minimizes overall traffic congestion.



**Pseudocode for Traffic Light Optimization Algorithm:**

function OptimizeTrafficLights(intersections, traffic\_data):

if all intersections optimized:

return current\_configuration

for each light configuration in possible\_configurations:

if configuration is valid:

set lights to configuration

optimize remaining intersections

return best\_configuration

**Implementation:**

def optimize\_traffic\_lights(intersections, traffic\_data):

def backtrack(current\_configuration):

if all\_optimized(current\_configuration):

return current\_configuration

for config in possible\_configurations():

if is\_valid(config, current\_configuration):

set\_lights(config)

result = backtrack(current\_configuration + [config])

if result:

return result

return None

return backtrack([])

### **Task 2: Simulate the Algorithm**

**Simulation Results:**

* Impact on Traffic Flow: Measure the reduction in congestion and travel times compared to a fixed-time system.
* Adaptability: Observe how the system adapts to changing traffic conditions.

### **Task 3: Compare with Fixed-Time System**

**Comparison with Fixed-Time System:**

* Backtracking Algorithm: Adapts to real-time conditions, optimizing traffic flow dynamically.
* Fixed-Time System: Uses predetermined timings that do not adapt to real-time changes, potentially leading to inefficiencies.

**Reasoning:**

The backtracking algorithm is effective for real-time traffic management due to its adaptability and optimization capabilities. In contrast, fixed-time systems are less responsive and may not optimize traffic flow effectively.