School of Engineering and Applied Science (SEAS), Ahmedabad University

CSE400: Fundamentals of Probability in Computing

Group Name: ITS - 3
Team Members:

- Even Patel (AU2340241)
- Dhyey Patel (AU2340245)
- Nishit Patel (AU2340261)
- Radha Patel (AU2340132)

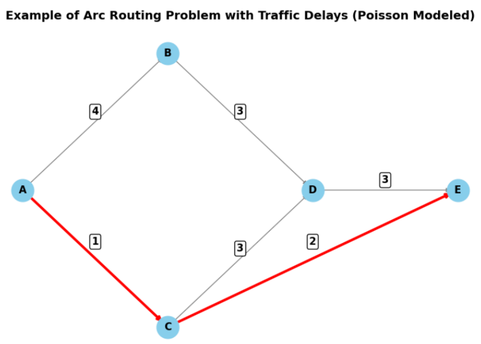
I. Background and Motivation

Background

- Arc Routing Problems (ARPs) are concerned with determining best routes to cover some of the edges in a network that often connect roads, pipelines, or service lines that need to be serviced. ARPs are important in everyday operations such as refuse collection, newspaper distribution, snow removal, and meter reading. In contrast to traditional routing problems that involve visiting nodes, ARPs are interested in serving particular arcs or streets, usually subject to limitations such as vehicle capacity or service time. Variants of ARPs, such as the Capacitated Arc Routing Problem (CARP) and the Chinese Postman Problem (CPP), have received extensive research interest because of their applicability and computational difficulties.

Motivation

- In our day-to-day lives, when we are going from one location to another, not only the distance, but the traffic situation also has a huge influence. A distance that may seem short on a map could be longer if the road is packed. Therefore, having an idea of the physical distance between two locations isn't sufficient. We need to consider the number of cars present on the road as well, since heavier traffic implies greater delay.


Example of Arc Routing Problem with Traffic Delays (Poisson Modeled)
B
A
D
E
C
4
3
3
1
3
2

II. Application

1. Urban Logistics & Delivery Services
   - **E-commerce Delivery (e.g., Amazon, FedEx, UPS):** Optimizing delivery routes considering peak-hour traffic to reduce delivery times and fuel costs.
   - **Food Delivery (e.g., Uber Eats, DoorDash):** Ensuring timely food delivery while accounting for dynamic road conditions.
2. Ride-Sharing & Public Transport Scheduling
   - **Taxi & Ride-Sharing (e.g., Uber, Lyft):** Optimizing driver assignment and routing based on real-time traffic data.
   - **Public Transport (e.g., Bus Scheduling):** Adjusting schedules dynamically to accommodate changing road congestion levels.
3. Emergency & Healthcare Services
   - **Ambulance & Emergency Response (e.g., 911 dispatch systems):** Ensuring the fastest routes are chosen dynamically to minimize response time.
   - **Mobile Healthcare Services:** Optimizing visits to patients in home-care scenarios, avoiding congested areas.
4. Waste Collection & Utility Services
   - **Garbage Collection Routes:** Planning collection paths to avoid heavy traffic periods.
   - **Utility Maintenance Crews:** Scheduling repair teams efficiently to minimize downtime and service delays.
5. Smart Cities & Traffic Management
   - **Adaptive Traffic Control Systems:** Integrating TDVRP with IoT-based traffic signals to dynamically adjust light timings.
   - **Dynamic Toll Pricing Systems:** Adjusting toll fees based on traffic conditions to incentivize better route choices.
6. Supply Chain & Freight Logistics
   - **Warehouse-to-Retailer Distribution:** Ensuring timely product shipments by considering traffic congestion patterns.
   - **Intermodal Transport Coordination:** Synchronizing truck and rail shipments efficiently under dynamic road conditions.

III. T1 - Mathematical Modelling and Mathematical Analysis

A. Modelling from Paper

The **Time-Dependent Capacitated Arc Routing Problem (TDCARP)** is defined on a graph $G = (V, E, A)$, where edges E_R require service and arcs A_R represent travel paths. Each service $u \in E_R \cup A_R$ has a demand q_u and a mode set M_u. The problem minimizes the total route duration while ensuring constraints:

1. **Route Constraints:** Each route starts/ends at depot 0, with demand q_u not exceeding vehicle capacity Q, and maximum route duration D.
2. **Speed Model:** Time-dependent speed functions $v_{ij}(t)$ and $\hat{v}ij(t)$ define travel and service speeds, leading to arrival time calculations $\Phi ij(t_i)$ and $\hat{\Phi}_{ij}(t_i)$ via integral equations.

Table 1: **Symbol Importance**

Symbol	Meaning
$\Phi_{ij}(t_i)$	Arrival time at node j when traveling from i at time t_i
$\hat{\Phi}_{ij}(t_i)$	Arrival time at node j when servicing edge (i, j) from t_i
$\Phi_{ij}^{-1}(t_j)$	Latest departure time from i to reach j at t_j (travel case)
$\hat{\Phi}_{ij}^{-1}(t_j)$	Latest departure time from i to reach j at t_j (service case)
$v_{ij}(t)$	Speed function for travel on edge (i, j) at time t
$\hat{v}_{ij}(t)$	Speed function when servicing edge (i, j) at time t
d_{ij}	Distance of edge (i, j)
t_i	Departure time from node i
t_j	Arrival time at node j
x	Integration variable representing possible arrival time at j
$T(i_\ell, j_\ell)$	The time taken for service (i_ℓ, j_ℓ)
$\hat{\Phi} i\ell j_\ell$	The transition function determining the time for a service
$\Psi_{j_{\ell-1} i_\ell}$	Updates the time based on the previous service
Φ_r	The total duration of route r
λ_r	A binary variable indicating whether route r is used
m	The total number of available vehicles
a_{ij}^r	A binary parameter indicating whether route r performs service (i, j)

- These equations use **time-dependent integration** to model travel/service duration dynamically.
- The **arrival time functions** Φ_{ij} and $\hat{\Phi}_{ij}$ are derived by integrating speed over time.
- The **inverse functions** $\Phi^{-1} ij$ and $\hat{\Phi}^{-1} ij$ are obtained by fixing arrival time and solving for departure time.

Table 2: **Importance of Equations**

Equation	**Description**
$\Phi_{ij}(t_i) = \left\{ x \,\middle\|\, \int_{t_i}^{x} v_{ij}(t)\,dt = d_{ij} \right\}$	Determines arrival time at node j from i based on time-dependent speed $v_{ij}(t)$.
$\hat{\Phi}ij(t_i) = \{x \mid \int t_i{}^x \hat{v}ij(t)\,dt = dij\}$	Calculates arrival time when servicing edge (i,j), considering service speed $\hat{v}_{ij}(t)$.
$\Phi^{-1}ij(t_j) = \{t_i \mid \int t_i{}^{t_j} v_{ij}(t)\,dt = d_{ij}\}$	Finds latest departure time from i to ensure arrival at j at t_j (travel case).
$\hat{\Phi}^{-1}ij(t_j) = \{t_i \mid \int t_i{}^{t_j} \hat{v}ij(t)\,dt = dij\}$	Finds latest departure time for servicing edge (i,j) to reach j at t_j.

Property 1. Functions Φ_{ij} are piecewise linear, continuous, and monotonic.

Property 2. Let $t_1, \ldots, t_{h_{ij}-1}$ be the breakpoints of the function v_{ij}. The function Φ_{ij} has up to $2(h_{ij}-1)$ breakpoints with values: $t_1, \ldots, t_k$,

$$\Phi^{-1}ij(t_l), \ldots, \Phi^{-1}ij(t_{h_{ij}-1})$$

where: $k = \arg\max\{x \mid t_x \leq \Phi_{ij}^{-1}(D)\}, \quad l = \arg\min\{x \mid t_x \geq \Phi_{ij}(0)\}$

Hence, it states that the arrival time function $\Phi_{ij}(t)$is given by:

$$\boxed{\Phi_{ij}(t) = V_{ij}^{-1}(V_{ij}(t) + d_{ij})}$$

Where:

- $V_{ij}(x) = \int_0^x v_{ij}(t)\,dt$ is the cumulative travel function (distance traveled up to x).
- V_{ij}^{-1} is the inverse function of V_{ij}.
- d_{ij} is the total travel distance between i and j.

Key Observations:

1. Instead of **directly integrating the speed function**, we use the **cumulative travel function** $V_{ij}(t)$.
2. The inverse function V_{ij}^{-1} allows us to compute the **arrival time efficiently**.
3. The approach simplifies time-dependent shortest path algorithms by using precomputed functions.

Table 3: **Importance of Time Propagation and Objective Function**

Equation	**Description**
$\begin{cases} T(i_\ell, j_\ell) = \hat{\Phi}i\ell j_\ell\left(\Psi_{j_{\ell-1}i_\ell}\left(T(i_{\ell-1}, j_{\ell-1})\right)\right), \\ T(i_1, j_1) = \hat{\Phi}i_1 j_1\left(\Psi 0 i_1(0)\right) \end{cases}$	Defines the time propagation for each service in a route.
$\min_{r\in\Omega} \sum_{r\in\Omega} \Phi_r \lambda_r$	The objective function minimizes the total route duration.

Table 4: **Route Selection Constraints and Their Importance**

Constraint Equation	Importance
$\sum_{r\in\Omega} \lambda_r = m$	**Fleet Size Constraint:** Ensures exactly m routes are selected for dispatch. This controls the number of vehicles used.
$\sum_{r\in\Omega} a_{ij}^r \lambda_r = 1,$ $\forall(i,j) \in E_R \cup A_R$	**Service Coverage Constraint:** Guarantees that each required arc or edge is serviced exactly once across all selected routes.
$\lambda_r \in \{0,1\}, \quad \forall r \in \Omega$	**Binary Route Selection:** Enforces that each route is either chosen (1) or not chosen (0), ensuring a discrete decision for each route.

B. Modelling by Group

Table 5: **Symbol Importance**

Symbol	Meaning
V	Set of nodes (intersections or locations)
E	Set of directed edges (roads)
w	Weight function representing travel time on each edge
$w_0(u,v)$	Number of vehicles on the road segment
α	Traffic impact factor (congestion coefficient)
λ	Vehicle arrival rate (vehicles per minute)
T	Time period in minutes
k	The number of vehicles
e	Euler's number
$d(v)$	shortest time from s to v
$P(v)$	shortest path from s to v

1. Graph Representation

The road network is modeled as a **directed weighted graph**:

$$\boxed{G = (V, E, w)}$$

Each edge $(u,v) \in E$ has a **base travel time** $w_0(u,v)$, which is modified based on traffic.

2. Traffic-Dependent Travel Time

Each edge (u,v) has a **time-dependent weight** based on traffic density:

$$\boxed{w(u,v) = w_0(u,v) \cdot (1 + \alpha \cdot N)}$$

The traffic density N is modeled as a **Poisson random variable**:

$$\boxed{N \sim \text{Poisson}(\lambda T)}$$

Poisson Distribution for Vehicle Counts:
The number of vehicles N on an edge follows a Poisson distribution:

$$\boxed{P(N = k) = \frac{(\lambda T)^k e^{-\lambda T}}{k!}, \quad k = 0, 1, 2, \ldots}$$

This Poisson-distributed traffic count N affects travel time as:

$$\boxed{w(u, v) = w_0(u, v)(1 + \alpha N)}$$

where α is a congestion factor.
The value of α is chosen based on congestion conditions:

$$\boxed{\alpha = \begin{cases} \text{Uniform}(0.1, 0.5), & \text{if } N < 5 \quad \text{(Light traffic)} \\ \text{Uniform}(0.5, 1.0), & \text{if } 5 \leq N \leq 10 \quad \text{(Moderate traffic)} \\ \text{Uniform}(1.0, 2.0), & \text{if } N > 10 \quad \text{(Heavy congestion)} \end{cases}}$$

3. Shortest Path Computation (Dijkstra's Algorithm)
The goal is to find the shortest travel time from a **source node** s to all other nodes.

Dijkstra's Algorithm (Modified for Traffic)

(a) **Initialize:**

$$d(s) = 0, \quad d(v) = \infty \quad \forall v \neq s$$

(Set source distance to 0, all others to infinity.)

(b) **Priority Queue:** Select node u with the smallest $d(u)$.

(c) **Update Neighbors:** For each neighbor v of u:

$$d(v) = \min(d(v),\ d(u) + w(u, v))$$

(d) **Repeat** until all nodes are processed.

4. Path Selection & Visualization

- Before and after updating **traffic conditions**, the algorithm finds the best route.
- If the **shortest path changes**, it indicates that **congestion affects routing**.

$$\boxed{P_{\text{before}}(C) \neq P_{\text{after}}(C) \Rightarrow \text{Traffic affects shortest path}}$$

IV. T2 - Code (with description of each line)

Implementation of Shortest Path in Dijkstra's Algorithm Using Poisson Distribution

Brief Idea of Overall Code

- The goal is to find the quickest path from node A to node E in a directed graph where each edge has a base travel time.
- Initially, the number of vehicles on paths A $\rightarrow$ B and A $\rightarrow$ C is given. This affects actual travel time due to congestion.
- Travel time is adjusted using:

$$\text{adjusted\_time} = \text{base\_time} \times (1 + \alpha \times \text{number\_of\_vehicles})$$

  where α is a congestion factor.
- New traffic conditions are simulated using a Poisson distribution to generate the number of arriving vehicles over time.
- Based on updated vehicle counts and α, new adjusted travel times are computed.
- Dijkstra's algorithm is used to find the quickest path from A to E before and after the traffic update.
- This enables comparison of route selection and total travel time under varying traffic conditions.

A. Importing Necessary Libraries

The required libraries are imported at the beginning of the script:

```python
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats
import poisson
```

These libraries help in graph creation, visualization, numerical computations, and traffic simulation.

B. Graph Initialization and Edge Definition

A directed, weighted graph is created using the `networkx` library. The nodes represent locations, and the edges represent roads with predefined travel times.

```python
def create_graph():
    G = nx.DiGraph()
    G.add_weighted_edges_from([
        ("A", "B", 5),
        ("A", "C", 5),
        ("B", "D", 2),
        ("D", "E", 1),
        ("C", "F", 8),
        ("F", "E", 1)
    ])
    return G
```

Listing 1: Graph Construction

C. Adjusting Travel Time Based on Traffic

Here, The formula used for travel time adjustment is:

$$\text{new time} = \text{base time} \times (1 + \alpha \times \text{number of vehicles})$$

where α is a congestion factor determined based on traffic density.

```
def adjust_travel_time(base_time, num_vehicles, alpha):
    density = num_vehicles
    return base_time * (1 + alpha * density)
```

Listing 2: Adjust Travel Time

D. Dijkstra's Algorithm with traffic handling

Dijkstra's Algorithm is a graph-based shortest path algorithm that efficiently finds the minimum distance from a given source node to other nodes in a weighted graph. In this implementation, the function `dijkstra(G, source)` leverages NetworkX's built-in methods: `single_source_dijkstra_path_length` to compute the shortest path distances and `single_source_dijkstra_path` to extract the corresponding shortest paths.

```
def dijkstra(G, source):
    distances = nx.single_source_dijkstra_path_length(G, source)
    paths = nx.single_source_dijkstra_path(G, source)
    return distances, paths
```

Listing 3: Dijkstra Algorithm

E. Graph Visualization

The following function visualizes a graph using NetworkX, ensuring that nodes and edges are clearly labeled. To enhance clarity, the shortest path is distinctly highlighted in red. Finally, the visualization is rendered and displayed using plt.show().

```
def visualize_graph(G, paths, adjusted_weights, title):
    plt.figure(figsize=(8, 6))
    pos = nx.spring_layout(G, seed=42)

    nx.draw(G, pos, with_labels=True, node_size=1000, node_color='
        lightblue', font_size=12, edge_color='gray')

```

```python
    for node in paths:
        if node != "A":
            path_edges = list(zip(paths[node], paths[node][1:]))
            nx.draw_networkx_edges(G, pos, edgelist=path_edges, width=3, edge_color="red")

    edge_labels = {edge: f"{adjusted_weights[edge]:.1f} min" for edge in G.edges}
    for edge, label in edge_labels.items():
        x, y = (pos[edge[0]][0] + pos[edge[1]][0]) / 2, (pos[edge[0]][1] + pos[edge[1]][1]) / 2
        plt.text(x, y, label, fontsize=10, color='black', bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

    plt.title(title)
    plt.legend(["Quickest Path"], loc="upper left")
    plt.show()
```

Listing 4: Graph Visualization

F. Updating Edge Weights

This function updates the weight of a specific edge (u, v) in the graph by modifying its 'weight' attribute, allowing dynamic adjustments to the graph structure.

```python
def update_edge_weight(G, u, v, new_weight):
    G[u][v]['weight'] = new_weight
```

Listing 5: Update Edge Weights

G. Simulation of Traffic Conditions

These parameters define traffic conditions, where lambda rate represents the average vehicle arrival rate, time period sets the simulation duration, and base time determines the base travel time per edge

```python
G = create_graph()

lambda_rate = 1.5
time_period = 5
base_time = 3
initial_vehicles_A_C = 6
initial_vehicles_A_B = 4
```

Listing 6: Traffic Parameters

H. Traffic Impact Calculation

The function determine alpha assigns a congestion factor α based on vehicle count, simulating light, moderate, or heavy traffic conditions. This factor is then used to compute the adjusted travel time, which updates the edge weights in the graph to reflect real-time traffic effects.

```python
def determine_alpha(vehicles):
    if vehicles < 5:
        return np.random.uniform(0.1, 0.5)
    elif 5 <= vehicles <= 10:
        return np.random.uniform(0.5, 1.0)
    else:
        return np.random.uniform(1.0, 2.0)
```

Listing 7: Determine Congestion Factor

The adjusted travel time is calculated:

```python
alpha_A_C = determine_alpha(initial_vehicles_A_C)
alpha_A_B = determine_alpha(initial_vehicles_A_B)

initial_time_A_C = adjust_travel_time(base_time, initial_vehicles_A_C, alpha_A_C)
initial_time_A_B = adjust_travel_time(base_time, initial_vehicles_A_B, alpha_A_B)
update_edge_weight(G, "A", "C", initial_time_A_C)
update_edge_weight(G, "A", "B", initial_time_A_B)
```

Listing 8: Apply Traffic Effects

I. Shortest Path Before and After Traffic

Dijkstra's algorithm is first executed on the initial graph to find the shortest path before traffic impact.

```python
distances_no_traffic, paths_no_traffic = dijkstra(G, "A")

print("\n Before Traffic Impact")
print(f"Traffic on A   C & A   B: {initial_vehicles} vehicles, time: {adjusted_time:.1f} min,  : {alpha:.2f}")
print(f"Chosen path before traffic: {'   '.join(paths_no_traffic['E'])}")
print(f"Total time for chosen path before traffic: {distances_no_traffic['E']:.1f} min")

visualize_graph
distances_no_traffic, paths_no_traffic = dijkstra(G, "A")
chosen_path_before = paths_no_traffic.get("E", [])
total_time_before = sum(G[u][v]['weight'] for u, v in zip(chosen_path_before, chosen_path_before[1:]))

print(f"\n Before Traffic Impact")
print(f"Initially\nTraffic on A   C path: {initial_vehicles_A_C} vehicles, time: {initial_time_A_C:.1f} min,  : {alpha_A_C:.2f}")
print(f"Traffic on A   B path: {initial_vehicles_A_B} vehicles, time: {initial_time_A_B:.1f} min,  : {alpha_A_B:.2f}")
print(f"Chosen Path: {'   '.join(chosen_path_before)}, Total Time: {total_time_before:.1f} min")

visualize_graph(G, paths_no_traffic, nx.get_edge_attributes(G, 'weight'), "Before Traffic: A   C is Shortest")
```

Listing 9: Run Dijkstra

Before traffic impact, Dijkstra's algorithm finds the shortest path and displays initial travel times. visualize graph highlights the best paths.

J. Simulation of New Traffic Conditions

The simulation introduces new traffic conditions by generating a random vehicle count using a Poisson distribution. The congestion factor α is updated, and edge weights are adjusted accordingly before rerunning Dijkstra's algorithm to compute the shortest paths.

```
new_vehicles_A_C = np.random.poisson(lambda_rate * time_period)
new_vehicles_A_B = np.random.poisson(lambda_rate * time_period)

alpha_A_C = determine_alpha(new_vehicles_A_C)
alpha_A_B = determine_alpha(new_vehicles_A_B)

updated_time_A_C = adjust_travel_time(base_time, new_vehicles_A_C, alpha_A_C)
updated_time_A_B = adjust_travel_time(base_time, new_vehicles_A_B, alpha_A_B)
update_edge_weight(G, "A", "C", updated_time_A_C)
update_edge_weight(G, "A", "B", updated_time_A_B)
```

Listing 10: Traffic Simulation with Updates

K. Running Dijkstra with New Traffic

After simulating new traffic conditions, we run Dijkstra's algorithm again to determine the shortest path with updated travel times.

```
distances_with_traffic, paths_with_traffic = dijkstra(G, "A")
chosen_path_after = paths_with_traffic.get("E", [])
total_time_after = sum(G[u][v]['weight'] for u, v in zip(chosen_path_after, chosen_path_after[1:]))

print(f"\n After Traffic Impact")
print(f"Traffic on A    C path: {new_vehicles_A_C} vehicles, time: {updated_time_A_C:.1f} min,   : {alpha_A_C:.2f}")
print(f"Traffic on A    B path: {new_vehicles_A_B} vehicles, time: {updated_time_A_B:.1f} min,   : {alpha_A_B:.2f}")
print(f"Chosen Path: {'    '.join(chosen_path_after)}, Total Time: {total_time_after:.1f} min")
```

Listing 11: Running Dijkstra with new traffic

L. Comparing Paths Before and After Traffic

Here, we compare the shortest paths before and after the impact of new traffic conditions to observe if the optimal route has changed.

```
if chosen_path_before == chosen_path_after:
    print("\n Route remains the same: A      E is still shortest.")
else:
    print("\n Route changed! New shortest path is:", "      ".join(
        chosen_path_after))
```

Listing 12: Comparing shortest paths

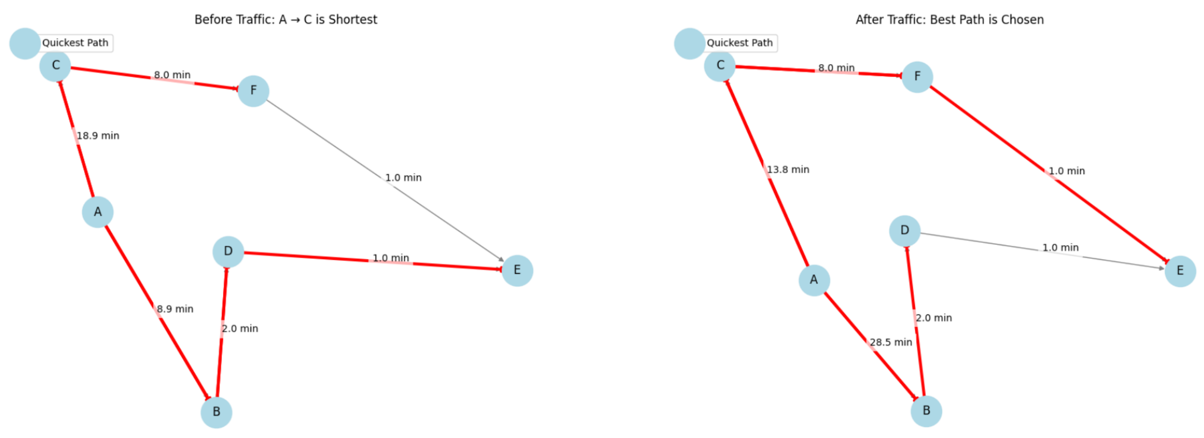
M. Visualizing the Updated Path

Finally, we visualize the graph again, highlighting the best path after considering the impact of new traffic conditions.

```
visualize_graph(G, paths_with_traffic, nx.get_edge_attributes(G, 'weight')
    , "After Traffic: Best Path is Chosen")
```

Listing 13: Visualizing updated traffic conditions

Simulation & Output Analysis of above Code



(a) Before Traffic Impact

(b) After Traffic Impact

Figure 1: Before and After Traffic Simulation Output

Case 1: Before Traffic

- Shortest path: $A \to B \to D \to E$
- Travel times: A → B: 8.9 min, B → D: 2.0 min, D → E: 1.0 min
- Total time: $8.9 + 2.0 + 1.0 = 11.9$ min
- The alternative path A → C → F → E has a total travel time of: A → C: 18.9 min, C → F: 8.0 min, F → E: 1.0 min Total: 27.9 min, which is much longer than 11.9 min.
- Thus, A → B → D → E is the optimal path before traffic.

Case 2: After Traffic

- Shortest path: $A \to C \to F \to E$
- Updated travel times: A $\to$ B: increased from 8.9 min to 28.5 min. A $\to$ C: decreased from 18.9 min to 13.8 min
- Travel times along new shortest path: A $\to$ C: 13.8 min, C $\to$ F: 8.0 min, F $\to$ E: 1.0 min. Total time: $13.8 + 8.0 + 1.0 = 22.8$ min
- The previous shortest path $A \to B \to D \to E$ now has: A $\to$ B: 28.5 min, B $\to$ D: 2.0 min, D $\to$ E: 1.0 min. Total: 31.5 min, which is longer than 22.8 min.
- Thus, the path $A \to C \to F \to E$ is now the optimal path after traffic.

Randomized Code Implementation

Dijkstra's Algorithm with Randomized Path Selection

- This is a variation of Dijkstra's shortest path algorithm.
- When multiple shortest paths with the same total weight exist, one is selected at random.
- The algorithm does not store or compare all equally optimal paths.
- It does not keep track of the other path, even though it is equally efficient.
- For example, if both paths A→B and A→C have the same travel time, the algorithm picks one randomly.
- The other equally efficient paths are ignored.
- This introduces randomness in path selection, simulating real-world decision-making where only one route is chosen.

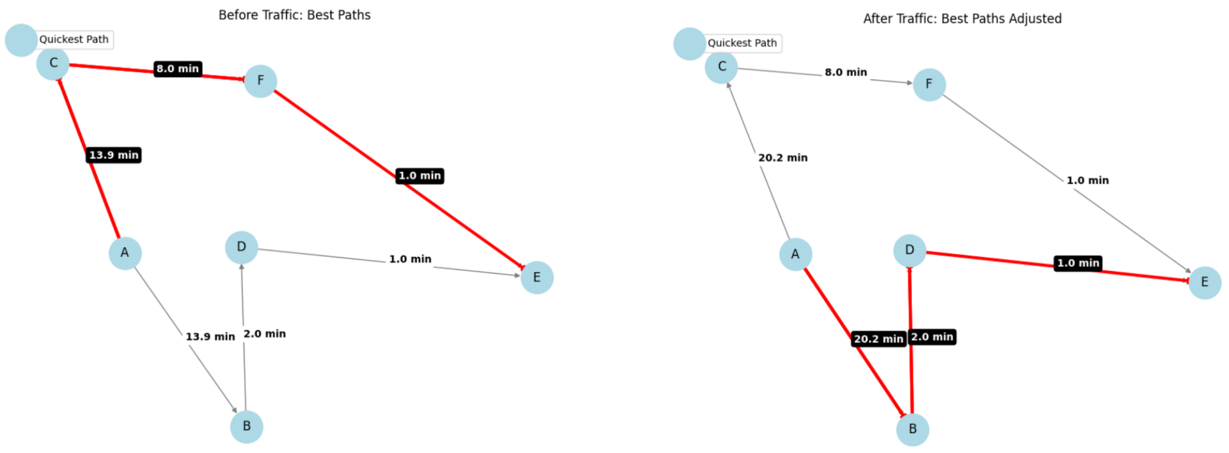
```python
def dijkstra(G, source):
    distances = {}
    paths = {}
    queue = [(0, source, [])]

    while queue:
        queue.sort()
        dist, node, path = queue.pop(0)

        if node in distances:
            continue

        distances[node] = dist
        paths[node] = path + [node]

        neighbors = list(G[node].items())
        neighbors.sort(key=lambda x: x[1]['weight'])

        min_weight = neighbors[0][1]['weight'] if neighbors else None
        min_neighbors = [n for n in neighbors if n[1]['weight'] ==
            min_weight]
        chosen_neighbor = random.choice(min_neighbors) if min_neighbors
            else None

        if chosen_neighbor:
            new_dist = dist + chosen_neighbor[1]['weight']
            queue.append((new_dist, chosen_neighbor[0], paths[node]))

    return distances, paths
```

Listing 14: Dijkstra Algorithm

- **Priority Queue Sorting:** The queue is sorted at each step to process the node with the smallest accumulated distance first, ensuring optimal path exploration.

- **Tracking Paths and Distances:** As nodes are processed, their shortest distance and path are stored to prevent revisits and guarantee correctness.
- **Randomized Edge Selection:** When multiple edges have the same minimum weight, one is randomly chosen, adding variability to path selection.

Simulation Output Analysis of Randomized Code



(a) Before Traffic Impact

(b) After Traffic Impact

Figure 2: Before and After Traffic Simulation Output

Before Traffic Impact

- The algorithm starts with base travel times and finds the shortest path from A to E.
- Since A $\rightarrow$ C (13.9 min) and A $\rightarrow$ B (13.9 min) have equal travel times, one is picked randomly.
- Here, it selects A $\rightarrow$ C, then moves through F (8.0 min) $\rightarrow$ E (1.0 min), Total travel time: $13.9 + 8.0 + 1.0 = 22.9$ min.

$$\text{A} \rightarrow \text{C (13.9 min)} \rightarrow \text{F (8.0 min)} \rightarrow \text{E (1.0 min)}$$

- The alternative path A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E is ignored. Total ignored path travel time: $13.9 + 2.0 + 1.0 = 16.9$ min.

$$\text{A} \rightarrow \text{B (13.9 min)} \rightarrow \text{D (2.0 min)} \rightarrow \text{E (1.0 min)}$$

- **The chosen path was A $\rightarrow$ C $\rightarrow$ F $\rightarrow$ E (22.9 min), while the ignored path A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E (16.9 min) was actually shorter.**

After Traffic Impact

- Traffic congestion increases, raising travel times.

- Now, A → C (20.2 min) becomes much longer than A → B (20.2 min), making the latter more likely to be chosen.
- This time, it selects A → B → D → E, Total travel time: $20.2 + 2.0 + 1.0 = 23.2$ min.

$$\text{A} \rightarrow \text{B (20.2 min)} \rightarrow \text{D (2.0 min)} \rightarrow \text{E (1.0 min)}$$

- The previously chosen path A → C → F → E is now ignored. Total ignored path travel time: $20.2 + 8.0 + 1.0 = 29.2$ min.

$$\text{A} \rightarrow \text{C (20.2 min)} \rightarrow \text{F (8.0 min)} \rightarrow \text{E (1.0 min)}$$

- The chosen path switched to A → B → D → E (23.2 min), while the ignored path A → C → F → E (29.2 min) was significantly longer.

Therefore, Dijkstra's algorithm randomly selects among equal-weight paths. Initially, it chose A → C → F → E, but after traffic changes, it switched to A → B → D → E.

V. T4 - Algorithm(Deterministic/Baseline and Randomized)

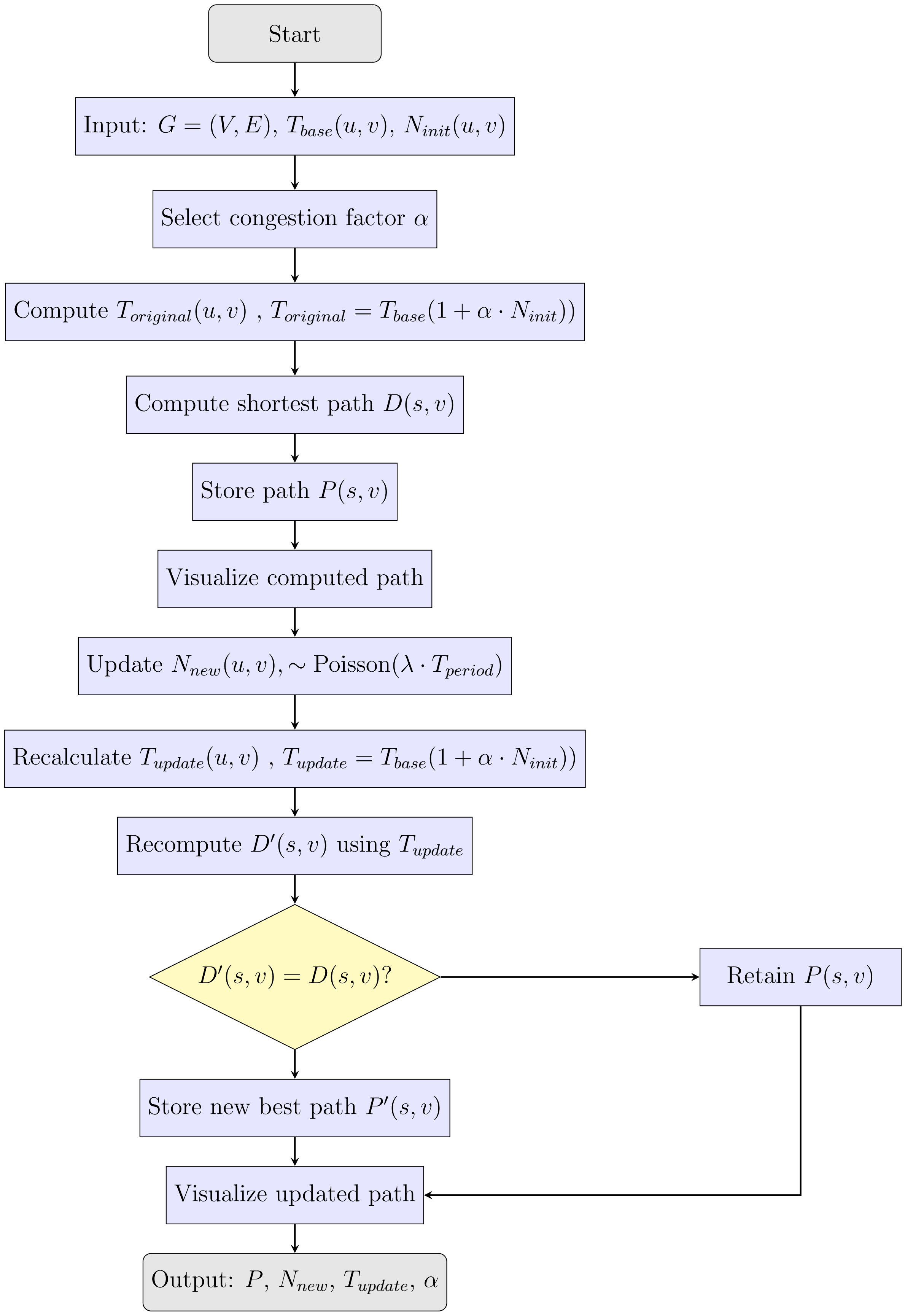
A. Deterministic Algorithm

A..1 Shortest Path in Dijkstra's algorithm using poisson distribution

Algorithm 1 Shortest Path in Dijkstra's algorithm using poisson distribution

```
Require: Graph $G = (V, E)$, base travel time $T_{\text{base}}(u, v)$, initial vehicle count $N_{\text{init}}(u, v)$,
    congestion factor $\alpha$.
Ensure: Optimal shortest path considering dynamic traffic conditions.
 1: Input: Graph structure, base travel times, and initial vehicle densities.
 2: Select congestion factor $\alpha$ based on traffic conditions:
      • $0 < \alpha < 1$                                   ▷ Normal traffic
      • $1 \le \alpha \le 2$                               ▷ Heavy congestion
      • $\alpha > 2$                                       ▷ Extreme congestion
 3: Compute original travel time for each edge:
      $$T_{\text{original}}(u, v) = T_{\text{base}}(u, v) \times (1 + \alpha \times N_{\text{init}}(u, v)) \quad (1)$$
 4: Compute shortest path $D(s, v)$ using Dijkstra's algorithm.
 5: Store the best path $P(s, v)$.
 6: Visualize the computed shortest path.
 7: Update vehicle count using Poisson distribution:
      $$N_{\text{new}}(u, v) \sim \text{Poisson}(\lambda \times T_{\text{period}}) \quad (2)$$
 8: Recalculate travel time:
      $$T_{\text{update}}(u, v) = T_{\text{base}}(u, v) \times (1 + \alpha \times N_{\text{new}}(u, v)) \quad (3)$$
 9: Recompute shortest path $D'(s, v)$ using $T_{\text{update}}(u, v)$.
10: if $D'(s, v) = D(s, v)$ then
11:     Path remains optimal, retain $P(s, v)$.
12: else
13:     Path updated, store new best path $P'(s, v)$.
14: end if
15: Visualize the updated shortest path.
16: Output: path, $N_{\text{new}}(u, v)$, $T_{\text{update}}$,$\alpha$
```

A..2 Flowchart of Shortest Path in Dijkstra's Algorithm using poisson distribution


Start
Input: $G = (V, E)$, $T_{base}(u, v)$, $N_{init}(u, v)$
Select congestion factor α
Compute $T_{original}(u, v)$, $T_{original} = T_{base}(1 + \alpha \cdot N_{init}))$
Compute shortest path $D(s, v)$
Store path $P(s, v)$
Visualize computed path
Update $N_{new}(u, v), \sim$ Poisson$(\lambda \cdot T_{period})$
Recalculate $T_{update}(u, v)$, $T_{update} = T_{base}(1 + \alpha \cdot N_{init}))$
Recompute $D'(s, v)$ using T_{update}
$D'(s, v) = D(s, v)$?
Retain $P(s, v)$
Store new best path $P'(s, v)$
Visualize updated path
Output: P, N_{new}, T_{update}, α

$\rightarrow$ This algorithm is based on Dijkstra's algorithm with Poisson distribution in number of vehicle counts, which shows deterministic behavior.

We first initialize the graph $G = (V, E)$ and take inputs as:

- T_{base}: Base travel time
- $N_{\text{init}}(u, v)$: Initial number of vehicles
- α: Traffic congestion factor

whew α is:

$$\begin{aligned} 0 < \alpha < 1 &\quad \text{Normal traffic} \\ 1 \leq \alpha < 2 &\quad \text{Heavy traffic} \\ \alpha \geq 2 &\quad \text{Extreme traffic} \end{aligned}$$

***Step-by-step Process**

1. After selection of α, compute the original travel time:

$$\boxed{T_{\text{original}}(u, v) = T_{\text{base}}(u, v) \cdot (1 + \alpha \cdot N_{\text{init}}(u, v))}$$

2. Compute shortest path using Dijkstra's algorithm, $D(S, V)$
3. From the computation, store the base path, $P(S, V)$
4. Then,through Poisson distribution to calculate new number of vehicles:

$$\boxed{N_{\text{new}}(u, v) \sim \text{Poisson}(\lambda \cdot T_{\text{period}})}$$

   where

   - λ = Average vehicle arrival rate
   - T_{period} = Time period for updating new vehicle counts

5. Calculate updated travel time:

$$\boxed{T_{\text{update}}(u, v) = T_{\text{base}}(u, v) \cdot (1 + \alpha \cdot N_{\text{new}}(u, v))}$$

6. Then we Recompute shortest path again using T_{update} i.e. updated travel time for $N_{\text{new}}(u, v)$ new vehicle counts
7. Further we check if new shortest path is equal to original path, if yes then original path is optimal otherwise we store new shortest path as P'(S, V) .

***Output**

- Path P
- $N_{\text{new}}(u, v)$
- $T_{\text{update}}(u, v)$
- α

***Remarks:** We also get simulation as output as seen in the result for deterministic case.

B. Randomized Algorithm

B..1 shortest path in Dijkstra's Algorithm with Randomized Path Selection

Difference between Deterministic and Randomized algorithm

$\rightarrow$ In this algorithm, instead of computing the shortest path directly through Dijkstra's Algorithm, we choose the neighbor randomly based on minimum distance.

$\rightarrow$ In the deterministic algorithm, we find the shortest path using Dijkstra's Algorithm where we only distribute the vehicular density using the Poisson distribution.

$\rightarrow$ In the randomised algorithm, we incorporate randomness not only in the vehicular density but also in the path-finding process itself by randomly choosing the neighboring node with minimum partial path cost. This introduces path variations, even when the travel time remains the same.

Explanation of randomization

$\rightarrow$ Here in randomised algorithm for the shortest path selection, we introduced randomness by initialising queue h with distance, node and path, where we first sort based on the minimum distance, then if the randomly chosen node if node exist,we return the path and distance

And if randomly chosen node is not present with the minimum path. Then we update its distance and path and again randomly choose the neighbouring node using,
Update distance:

$$\boxed{\text{distance[node]} = \text{dist}}$$

Update path:

$$\boxed{\text{path[node]} = \text{path} + \text{[node]}}$$

Compute new distance:

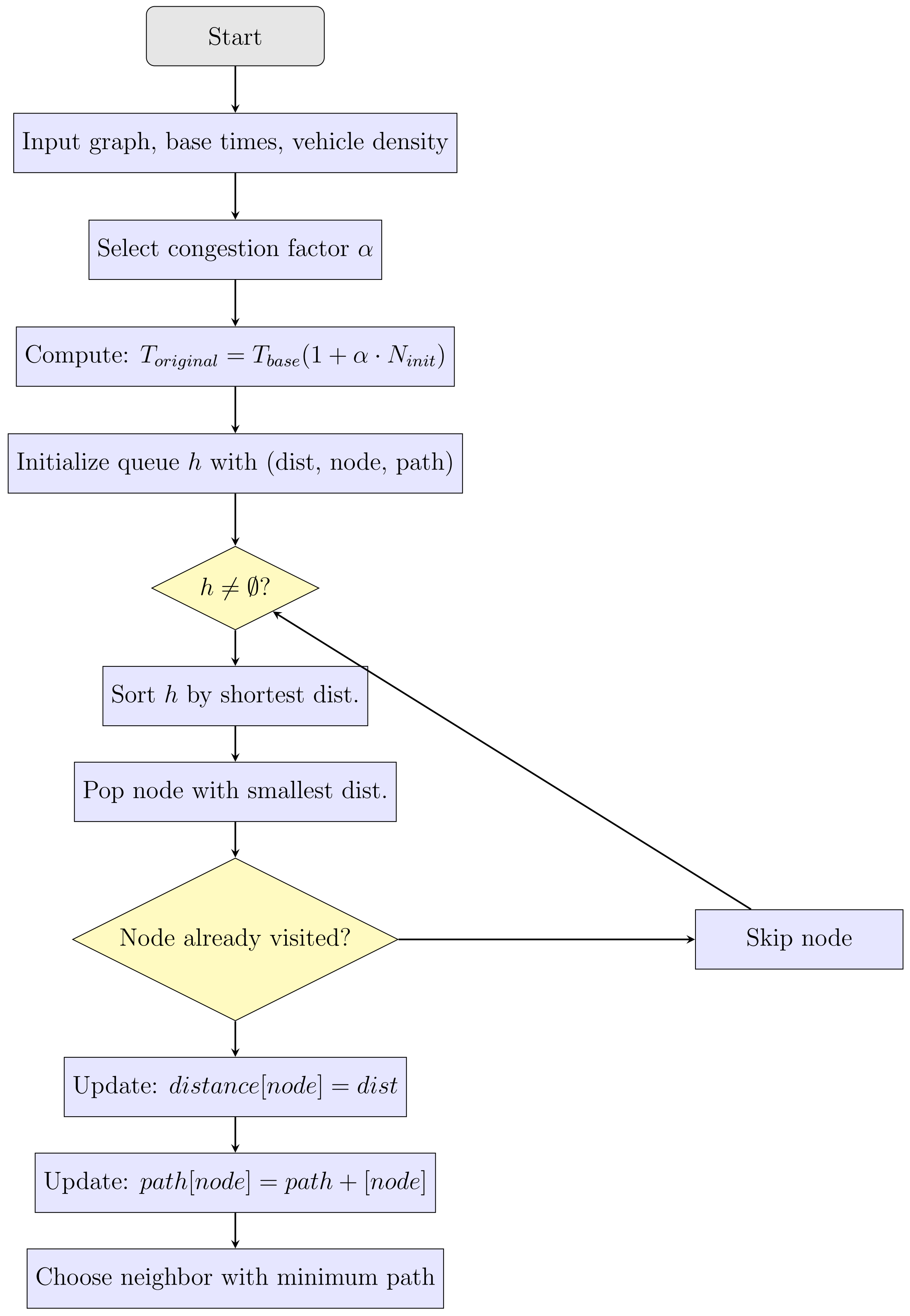
$$\boxed{\text{new distance[node]} = \text{dist} + \text{weight(path)}}$$

Whereas in deterministic algorithm we calculate shortest by Dijkstra's Algorithm and return the path and distance.

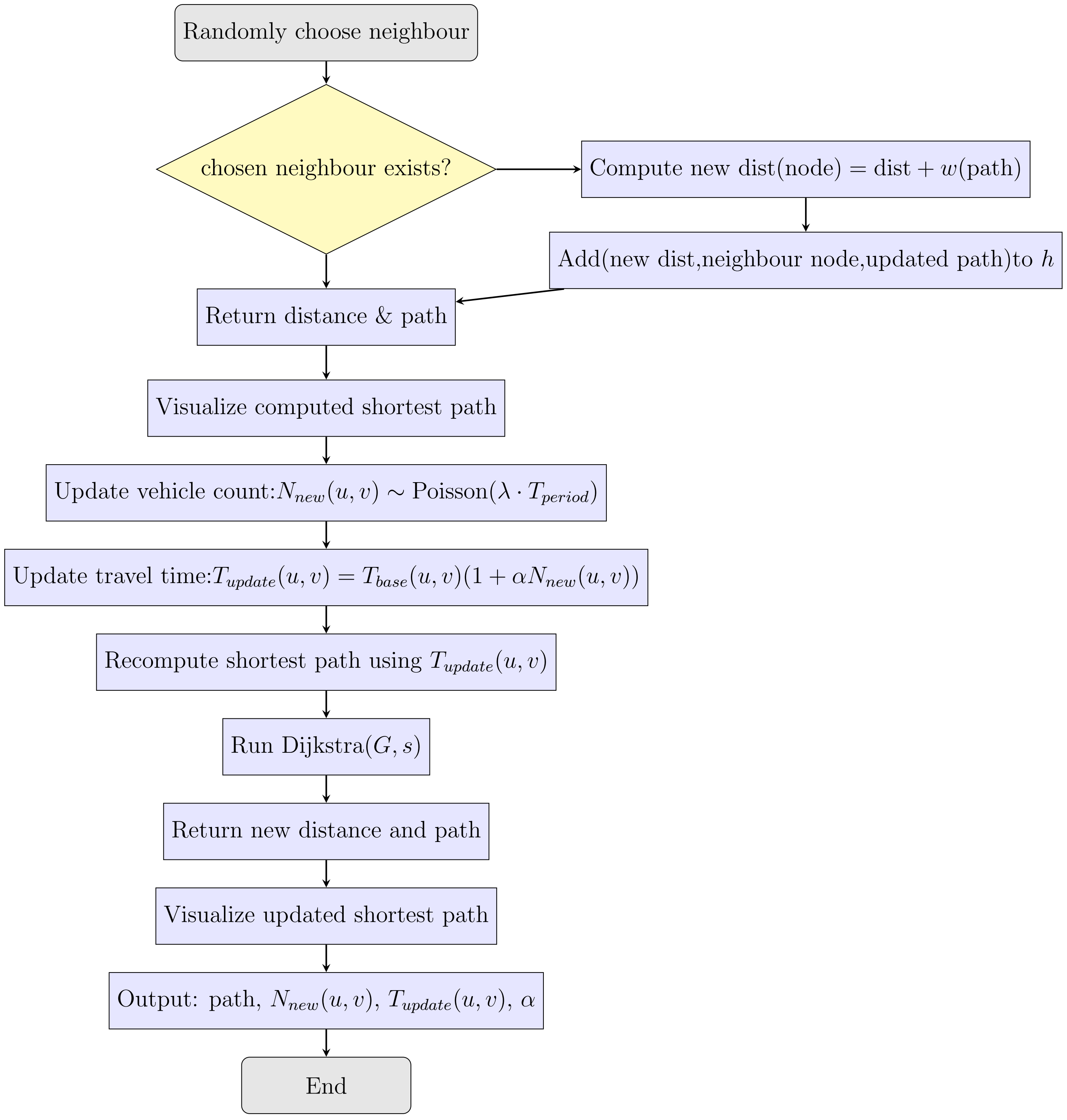
Algorithm 2 Shortest Path with Randomized Congestion Simulation

```
Require: Graph $G = (V, E)$, base travel time $T_{\text{base}}(u, v)$, initial vehicle count $N_{\text{init}}(u, v)$,
    congestion factor $\alpha$.
Ensure: Optimal shortest path considering dynamic traffic conditions.
 1: Input: Graph structure, base travel times, and initial vehicle densities.
 2: Select congestion factor $\alpha$ based on traffic conditions:
      • $0 < \alpha < 1$                                        ▷ Normal traffic
      • $1 \leq \alpha \leq 2$                                  ▷ Heavy congestion
      • $\alpha > 2$                                            ▷ Extreme congestion
 3: Compute original travel time for each edge:
      $T_{\text{original}}(u, v) = T_{\text{base}}(u, v) \times (1 + \alpha \times N_{\text{init}}(u, v))$    (4)
 4: Define Dijkstra's Algorithm: dijkstra(G, s)
 5: Initialize queue $h$ holding tuples (current distance, current node, path taken).
 6: while $h \neq \emptyset$ do
 7:     Sort $h$ based on shortest distance.
 8:     Pop element from $h$ with the smallest distance.
 9:     if Node is already visited then
10:         Skip.
11:     else
12:         Update distance:
              $\text{distance[node]} = \text{dist}$    (5)
13:         Update path:
              $\text{path[node]} = \text{path} + \text{[node]}$    (6)
14:         Randomly choose a neighbor based on minimum path.
15:         if chosen neighbor exists then
16:             Compute new distance:
                  $\text{new distance[node]} = \text{dist} + \text{weight(path)}$    (7)
17:             Add (new distance, neighbor node, updated path) to queue.
18:         end if
19:     end if
20:     return distance, path
21: end while
22: Visualize the computed shortest path.
23: Update vehicle count using Poisson distribution:
      $N_{\text{new}}(u, v) \sim \text{Poisson}(\lambda \times T_{\text{period}})$    (8)
24: Recalculate travel time:
      $T_{\text{update}}(u, v) = T_{\text{base}}(u, v) \times (1 + \alpha \times N_{\text{new}}(u, v))$    (9)
25: Compute new shortest path using updated travel times:
26: Define dijkstra(G, s).
27: return new distance, new path.
28: Visualize the updated shortest path.
29: Output: path, $N_{\text{new}}(u, v)$, $T_{\text{update}}$,$\alpha$
```

B..2 Flowchart of Shortest path in Dijkstra's Algorithm with Randomized Path Selection



Start
Input graph, base times, vehicle density
Select congestion factor α
Compute: $T_{original} = T_{base}(1 + \alpha \cdot N_{init})$
Initialize queue h with (dist, node, path)
$h \neq \emptyset$?
Sort h by shortest dist.
Pop node with smallest dist.
Node already visited?
Skip node
Update: $distance[node] = dist$
Update: $path[node] = path + [node]$
Choose neighbor with minimum path

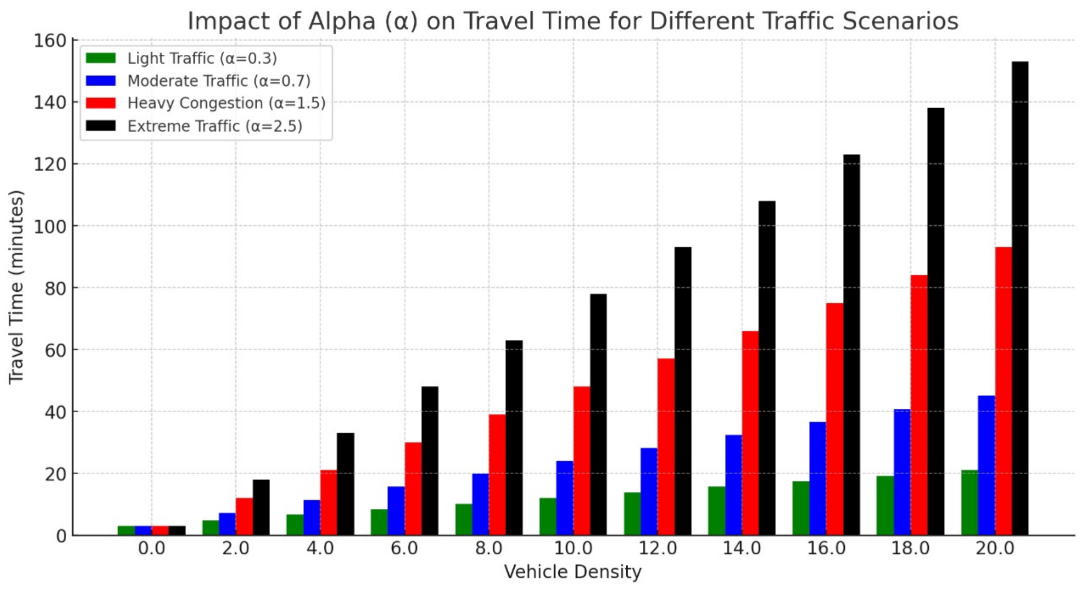

Randomly choose neighbour
chosen neighbour exists?
Compute new dist(node) = dist + w(path)
Add(new dist,neighbour node,updated path)to h
Return distance & path
Visualize computed shortest path
Update vehicle count:$N_{new}(u, v) \sim \text{Poisson}(\lambda \cdot T_{period})$
Update travel time:$T_{update}(u, v) = T_{base}(u, v)(1 + \alpha N_{new}(u, v))$
Recompute shortest path using $T_{update}(u, v)$
Run Dijkstra(G, s)
Return new distance and path
Visualize updated shortest path
Output: path, $N_{new}(u, v)$, $T_{update}(u, v)$, α
End

VI. T3 -Inferences (Domain + CS perspective)

A. CS Perspective

A..1 Understanding α (alpha):

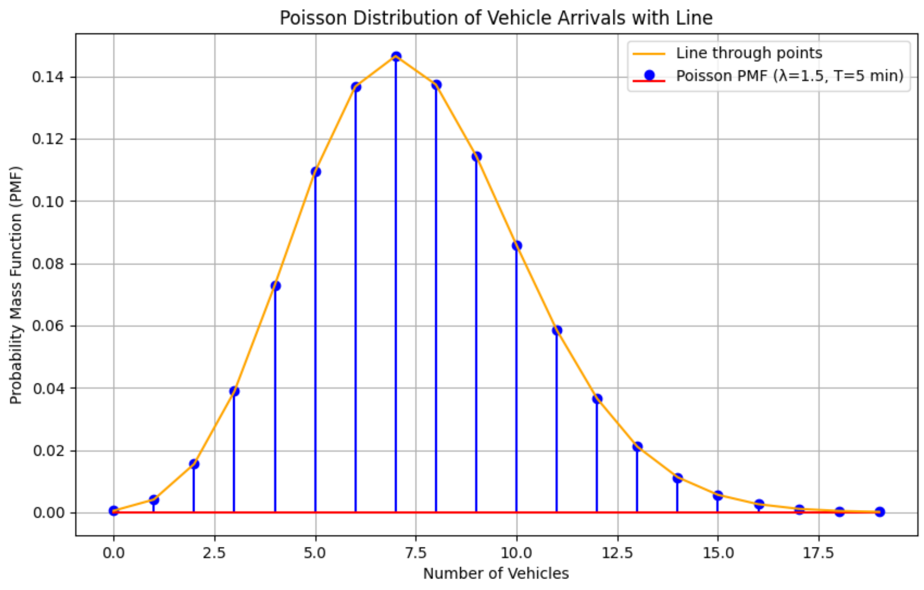
α is a parameter that quantifies how sensitive the travel time is to changes in traffic density. Essentially, it amplifies the effect of the number of vehicles on the base travel time.



Inferences Based on α Ranges:

- **Light Traffic** $(0.1 \leq \alpha \leq 0.5)$**:**
  Represented by the green bars in the plot. Travel time increases very slowly with vehicle density, indicating smooth flow and minimal congestion.
- **Moderate Traffic** $(0.5 < \alpha \leq 1.0)$**:**
  Shown using blue bars. Travel time grows more noticeably with density, but the increase remains manageable and relatively linear.
- **Heavy Congestion** $(1.0 < \alpha \leq 2.0)$**:**
  Represented by red bars. Travel time rises steeply as vehicle density increases, indicating the onset of significant congestion effects.
- **Extreme Traffic Conditions** $(\alpha > 2.0)$**:**
  Depicted in black. Travel time increases dramatically and non-linearly with density, suggesting highly unstable and congested traffic scenarios.

A..2 Poison Distribution



The Poisson distribution of vehicle arrivals shows a peak at approximately 7-8 vehicles, indicating the most probable congestion level. The probability mass function (PMF) follows a right-skewed trend, highlighting the decreasing likelihood of higher vehicle counts within the observed time period.

A..3 Time Complexity

1. **Shortest Path Code in Dijkstra's Algorithm Using Poisson Distribution**
   - **Best Case:** $O(V \log V)$
     - **Graph Type:** Sparse graphs where $E \approx V$ (e.g., tree-like structures).
     - **Reason:** Dijkstra's algorithm processes nodes efficiently with minimal priority queue operations due to fewer edges.
   - **Worst Case:** $O(V^2 \log V)$
     - **Graph Type:** Dense graphs where $E \approx V^2$ (e.g., fully connected graphs).
     - **Reason:** The priority queue grows large, and every node must be visited with many edge relaxations, leading to a higher computational cost.
   - **Average Case:** $O((V + E) \log V)$
     - **Graph Type:** Real-world graphs that are neither sparse nor dense (e.g., $E = kV$, where k is a constant).
     - **Reason:** Balanced priority queue operations result in a moderate computational cost.
2. **Randomized code (Randomization in Search Path Algorithm)**

- **Best-Case:** $O(V)$
  - Occurs in graphs where each node has at most one or two neighbors(**chain/tree-like graphs**) where the queue remains small, minimizing sorting operations.
- **Worst-Case:** $O(V^2 \log V)$
  - Occurs in dense graphs (e.g., nearly complete graphs) where most nodes are interconnected. The priority queue contains all V nodes at many steps, requiring frequent sorting
- **Average-Case Complexity:** $O((V + E)logV)$
  - Occurs in sparse graphs, where each node connects to only a few others. Sorting happens during insertion and deletion, but the queue size remains moderate.

B. Domain Inferences

B..1 Applications in Intelligent Transportation Systems (ITS)

Integrating a Poisson-based traffic model with dynamic travel time adjustments using the α parameter creates a forward-thinking framework for real-time traffic management. This approach empowers ITS to:

- **Adapt Instantly:** Continuously update routing recommendations based on live traffic conditions.
- **Enhance Safety:** Provide critical support for autonomous vehicles by anticipating and mitigating congestion.
- **Promote Sustainability:** Reduce fuel consumption and emissions through smoother traffic flows and optimized routing.

These capabilities work together to deliver a transportation network that is not only efficient but also responsive and eco-friendly.

B..2 Robustness and Flexibility in Route Selection

The introduction of randomized elements into the routing algorithm mirrors the unpredictable nature of real-world driving conditions. This design choice ensures that:

- **Exploration of Alternatives:** The system dynamically considers multiple paths, ensuring that unexpected incidents like accidents or sudden congestion can be bypassed.
- **Resilience Under Uncertainty:** By not locking into a single predetermined route, the network maintains efficiency even when conditions change abruptly.
- **Reliability in Navigation:** Users benefit from a system that adapts flexibly, thereby enhancing the overall robustness of route planning.

This flexible approach reinforces the system's capacity to handle diverse traffic scenarios while maintaining optimal performance.

B..3 Dynamic Routing Under Congestion

Dynamic routing is key to managing modern urban mobility. By recalculating travel times in real-time according to current traffic densities, the system is able to:

- **Respond Swiftly:** Immediately adjust travel routes as congestion builds up, minimizing delays.
- **Alleviate Bottlenecks:** Identify and suggest alternate pathways during peak periods, thereby reducing traffic congestion.
- **Boost Network Efficiency:** Enhance overall traffic flow and commuter satisfaction by continuously optimizing route selection.

This adaptive strategy is crucial for mitigating the negative impacts of urban congestion and ensuring smooth, efficient transportation throughout the network.

VII. T5 - Derivation of Bounds and Results (new inferences)

Probabilistic Bound for Shortest Path Selection

Let $G = (V, E)$ be a fully connected directed graph with $|V| = n$ nodes, where $n \geq 3$. Let s and t be the source and destination nodes respectively, with $s \neq t$.

Defining:

- T: Total number of simple (loop-free) paths from s to t
- S: Number of shortest paths from s to t

If each path is equally likely to be selected at random, then the probability of choosing a shortest path is:

$$\boxed{P = \frac{S}{T}} \quad \text{where } 0 < P \leq 1$$

Case 1: Random Weighted Graph (Typically Unique Shortest Path)

In a graph where edge weights are assigned randomly (e.g., uniformly), the shortest path is unique with high probability, i.e., $S = 1$.

The total number of simple paths from s to t is:

$$T = \sum_{k=1}^{n-2} \binom{n-2}{k} \cdot k!$$

Thus, the probability becomes:

$$\boxed{P = \frac{1}{\sum\limits_{k=1}^{n-2} \binom{n-2}{k} \cdot k!}}$$

Case 2: Multiple Shortest Paths of Equal Length L (e.g., Unweighted Graph)

If multiple shortest paths of length L exist, the number of such shortest paths is:

$$S = \binom{n-2}{L-1} \cdot (L-1)!$$

Then the probability becomes:

$$\boxed{P = \frac{\binom{n-2}{L-1} \cdot (L-1)!}{\sum\limits_{k=1}^{n-2} \binom{n-2}{k} \cdot k!}}$$

Conclusion

- The correct bound is the ratio $\frac{S}{T}$, which is always within $(0, 1]$.
- As the graph grows, T increases faster than S, making random selection of the shortest path less likely.
- This formulation ensures mathematical validity and better reflects path selection in randomized routing algorithms.
- This probabilistic bound provides a structured approach to analyze randomized path selection, especially under varying traffic conditions in large-scale networks.

VIII. References

Papers with Code, “Arc Routing with Time-Dependent Travel Times and Paths,” *Paperswithcode.com*, 2021. [Online]. Available: https://cs.paperswithcode.com/paper/arc-routing-with-time-dependent-travel-times [Accessed: Apr. 5, 2025].

in, “Total number of paths between two nodes in a complete graph,” *Mathematics Stack Exchange*, Aug. 26, 2017. [Online]. Available: https://math.stackexchange.com/questions/2406920/total-number-paths-between-two-nodes-in-a-complete-graph [Accessed: Apr. 5, 2025].

”Poisson Distribution: Learn Definition, Formula, Examples,” *DataCamp*. [Online]. Available: https://www.datacamp.com/tutorial/poisson-distribution [Accessed: Apr. 5, 2025].