Algorithms: Term Project: Savior of the Louvre

Objective:

It's a bright morning in Paris — the streets are buzzing, tourists crowd the cafés, and the city hums with energy. Suddenly, chaos erupts at the **Louvre Museum**. A group of burglars has just pulled off a daring **daytime art heist**!

They jump into their getaway car — **Car A** — and speed off through the sunlit streets,

Within minutes, the police alarm blares across the city. The Paris Police spring into action, racing to intercept the thieves in **Car B**. The chase is on!!

Your mission: **build a dynamic, real-time chase simulator** using a synthetic **directed weighted graph** to represent the streets of Paris. The system should simulate **changing traffic conditions, temporary blockages, and one-way restrictions**, requiring the cars to **recompute optimal paths in real time**.

The goal is to use and extend your understanding of **graph traversal algorithms** such as **Dijkstra's Algorithm**— with an applied, visual, and interactive twist.

Scenario Description:

- The city is represented by a **directed weighted graph** given as a JSON file:
  " graph_with_metadata"
  - **Nodes** represent intersections or checkpoints.
  - **Edges** represent roads connecting those intersections.
  - **Weights** on edges represent the **distance+traffic**

- **Car A** (Burglars) starts at a **source node - node 0** and tries to reach **the exit node, node 48** as fast as possible.

- **Car B** (Police) starts from the police station node (node **49**) but **after a delay of 3 timesteps**. Its goal is to **catch Car A** before it reaches the exit node.

- The environment is **dynamic**:
  - Traffic jams can increase edge weights temporarily.
  - Road blockages can close edges temporarily.
  - One-way enforcements

Core Functionalities (Required)

1. Graph Loading (City Map)

- Load the graph given in "graph_with_metadata"
  - The load_graph function is given below – use the adjacency dict, positions (co-ordinates of the nodes), metadata and exit_nodes for your coding purpose. You can use the graph object G_loaded only for visualization and animation purpose.

2. Dijkstra's Shortest Path

- Implement **Dijkstra's Algorithm from scratch** — no imported heroes this time. The core Dijkstra function should look like:
  def **dijkstra***(adjacency_dict, start_node, target_nodes)*

3. Two-Car Simulation

Initialize your heroes (or villains):

- **Car A → Starts immediately from start_node (node 0)with speed 1x .**
- **Car B → Joins the chase (from node 49) after a delay (delay = 3 time steps).**
- **Car B is faster — 1.5× the speed of Car A.**

4. Dynamic Events

- Generate dynamic events at each timestep with probability p = 0.3
- Traffic jams, roadblocks, or new one-way enforcements appear — forcing both drivers to adapt. When an event ends, the road reopens or clears, restoring the original map.
- Implement three types of random events that may occur at each timestep (with probability p = 0.3):

Event Type	Description	Effect	Duration
Traffic Jam	Increases weight of an edge	Multiply weight by 2.0–2.5	3 steps
Blockage	Temporarily removes an edge	Edge unavailable	4 steps
One-way	Removes reverse edge of a road	Direction becomes one-way	6 steps

- When events resolve, the original graph structure/weight is restored.

5. Dynamic Path Replanning

Below are the rules of the chase:

For car A:
Car A, burglar, begins at a given start node in the graph and tries to reach the exit node. It has full knowledge of the graph connectivity, the weights of the edges, and the locations of the exit, but it does not know anything about Car B's position or behaviour. At the start of the simulation, Car A computes the shortest path to the nearest exit and starts moving along that path at a constant speed of one unit per timestep. The edge weights represent the travel time or distance between two connected nodes, so Car A's progress along an edge accumulates each step until it equals or exceeds the weight of that edge. Once an edge is completed, the car moves to the next node on its current path and continues until it either reaches an exit or some change in the environment forces it to reconsider its route.

Whenever a dynamic event such as a traffic slowdown, a blockage, or a temporary one-way restriction occurs on the graph, Car A recomputes it path. If the event takes place while it is still mid-edge, it will continue until it reaches the next node before recomputing a new shortest path. If it happens to be exactly at a node when the event is introduced, it immediately recalculates a new path to the exit from its current position. Car A never recalculates unless such an event occurs; otherwise, it keeps following the same plan. The simulation for Car A ends as soon as it reaches the exit node.

For car B:

Car B, the police, starts from its own source node in the same graph but with different information and intent. Car B's goal is to intercept Car A before it reaches an exit first. Unlike Car A, Car B does not know which node is the exit node, but it always knows the current position of Car A. This visibility is perfect — if Car A is sitting at a node, Car B knows that node; if Car A is moving along an edge, Car B knows both endpoints and how far Car A has progressed along that edge. Car B remains stationary for the first three timesteps, simulating a delayed response, and then starts moving at a faster speed of 1.5 units per timestep. Once the delay is over, Car B computes the shortest path to Car A's current position. If Car A is on an edge, the destination is taken as the node that Car A is moving toward. Car B then begins to follow that path.

After every edge traversal or whenever a new dynamic event modifies the graph, Car B recalculates its shortest path again toward Car A's updated position. This means Car B's pursuit strategy continuously adapts to both the moving target and the changing road conditions. The two cars move once at each timestep — Car A makes the first move following its planned route to the exit, and Car B chasing it with updated knowledge of its location. A capture is declared if at any point both cars occupy the same node, or if they are on the same edge with less than 0.5 unit of distance between them.

Throughout the simulation, random dynamic events may occur that alter the state of the graph. A traffic event temporarily increases the travel time of an edge for 3 time_steps, a

blockage removes an edge completely for 4 time_steps, and a one-way event disables one direction of a bidirectional connection. These events are tracked in the list of active events and automatically revert after their duration expires, restoring the original connectivity. Whenever such an event is triggered, both cars are immediately aware of the new graph state and act according to their own recomputation rules.

The simulation proceeds step by step until one of three outcomes occurs: Car A successfully reaches an exit, in which case it escapes; Car B catches Car A, in which case it wins; or the maximum number of allowed timesteps is reached, in which case the simulation ends without a result.

Simulation Loop (Suggested Structure)

for step in range(max_steps=50):

1. Move Car A (After initial Dijkstra)
2. Move Car B (After initial Dijkstra/after re-computation at a node)
3. Check if caught or reached
4. Possibly trigger a random event
5. Resolve expired events
6. Recompute paths if environment changes
7. Save current state in the log file

6. Catch and Win Conditions

- **Car A wins** if it reaches the exit node before being caught.
- **Car B wins** if:
    - It occupies the same node as Car A at some point of time
    - Both cars are on the same edge and within a threshold of 0.5.

Sample outputs:

Car A reached exit 17 first! The burglars vanish like thin air...

Car B caught Car A at node 12! **Paris Police**!! **Saviour of the Louvre** !!!.

Expected Output Simulation Log:

At each time_step (max time_steps = 50), log the history of the simulation as formatted below and save them in a 'simulation.json' file-

```
history.append({
    "step": step,
    "carA": {
      "pos": carA["pos"],
      "edge_from": carA.get("edge_from"),
      "edge_to": carA.get("edge_to"),
      "progress": carA.get("progress", 0),
      "Dijkstra_path": list(carA["path"]) if carA.get("path") else [],
    },
    "carB": {
      "pos": carB["pos"],
      "edge_from": carB.get("edge_from"),
      "edge_to": carB.get("edge_to"),
      "progress": carB.get("progress", 0),
      "Dijkstra_path": list(carB["path"]) if carB.get("path") else [],
    },
    "caught":caught,
    "reached":reached,
    "log_events (Blockage, Traffic, One-way and between which nodes)":log_events
})
```

Visualization (Optional)

Use **matplotlib** or **networkx.draw()** to visualize the thrilling pursuit:

- Nodes (intersections), Directed edges (roads), Car A (Burglars), Car B (Police) Exit nodes (green).
- Optionally highlight edges when traffic or blockages occur.

```
import networkx as nx
import json
def load_graph(filename="graph_with_metadata.json"):
    with open(filename, "r") as f:
        data = json.load(f)

    # Convert adjacency dict keys to int and neighbor node IDs to int
    adjacency_raw = data["adjacency"]
    adjacency = {int(k): [(int(n), w) for n, w in v] for k, v in
adjacency_raw.items()}

    positions = {int(k): tuple(v) for k, v in
data["positions"].items()}
    metadata = data.get("metadata", {})
    end_nodes_loaded = metadata.get("exit_nodes", [])
    exit_nodes = end_nodes_loaded[1:]


    # Reconstruct directed graph
    G_loaded = nx.DiGraph()
    for node, neighbors in adjacency.items():
        for nbr, weight in neighbors:
            G_loaded.add_edge(node, nbr, weight=weight)

    print(f"Graph successfully reloaded from '{filename}'")
    print(f"Metadata: {metadata}")

    return G_loaded, adjacency, positions, metadata, exit_nodes
```