Hackathon Task: Fleet Management System with Traffic Negotiation for Multi-Robots

Objective:

Develop a visually intuitive and interactive Fleet Management System using a Python GUI, capable of managing multiple robots simultaneously navigating through an environment. Robots must negotiate traffic based on the provided navigation graph (nav_graph), avoiding collisions, dynamically assigning tasks, and clearly visualizing robot movements and statuses.

Detailed Task Description:

You will be provided with a navigation graph (`nav_graph.json`) describing vertices (locations) and lanes (connections between these locations). Your Fleet Management System should feature a graphical interface to visually display and interact with this environment, robots, and their navigation paths.

Your system must support the following features:

1. Visual Representation

- **Environment Visualization:**
    - Display all vertices (locations) and lanes clearly.
    - Clearly mark locations with name and intersections.
    - Vertices should be interactable (clickable) to spawn robots or assign tasks.
- **Robot Visualization:**
    - Robots must be visually distinct (different colors or icons).
    - Real-time visualization of robots moving along lanes towards their destinations.
    - Clearly indicate robot statuses (moving, waiting, charging, task complete).

2. Robot Spawning

- **Interactive GUI:**
    - Allow users to spawn robots dynamically by clicking on any vertex.
    - Each spawned robot must have a unique identifier displayed.

3. Navigation Task Assignment

- **Interactive Task Assignment:**
  - Click on a robot to select it, and click on a destination vertex to assign the navigation task visually.
  - Robots begin navigation immediately after task assignment.

4. Traffic Negotiation & Collision Avoidance

- **Real-Time Traffic Management:**
  - Robots must navigate efficiently, negotiating traffic by avoiding lane collisions.
  - Implement mechanisms for robots to wait or queue at occupied lanes or intersections.
  - Visualize waiting status clearly, indicating when robots are blocked.

5. Dynamic Interaction

- **Runtime Flexibility:**
  - Allow real-time spawning of new robots and assigning new tasks without interrupting currently navigating robots.

6. Occupancy and Conflict Notifications

- **User Alerts:**
  - Visually notify the user immediately when a requested path or vertex is occupied or blocked.
  - Provide clear pop-up messages or visual indicators.

7. Logging & Monitoring

- **Detailed Logging:**
  - Continuously log robot actions, path choices, waiting conditions, and task completions in a dedicated log file (`fleet_logs.txt`).
  - Optionally display real-time logs or status updates in the GUI for immediate feedback.

Understanding nav_graph:

Your provided nav_graph JSON includes:

- **Vertices:** Coordinates with attributes:
- `name`: A human-readable identifier.
- `is_charger` (optional): Indicates charging stations.
- Example:

```
[19.037718, -6.879006, {"name": "P1", "is_charger": true}]
```

- **Lanes:** Define paths between vertices:
- Example:

```
[0, 4]  // robots can move between vertices at index 0 and 4.
```

Recommended Project Structure:

```
fleet_management_system/
├── data/
│   └── nav_graph.json
├── src/
│   ├── models/
│   │   ├── nav_graph.py         # Parsing and representing nav_graph
│   │   └── robot.py             # Robot behaviors and attributes
│   ├── controllers/
│   │   ├── fleet_manager.py     # Robot task and state management
│   │   └── traffic_manager.py   # Traffic negotiation and collision
handling
│   ├── gui/
│   │   └── fleet_gui.py         # Interactive visualization
(Tkinter/PyQt/Pygame)
│   ├── utils/
│   │   └── helpers.py           # Supporting functions (path algorithms,
etc.)
│   ├── logs/
│   │   └── fleet_logs.txt
│   └── main.py                  # Application entry point
├── requirements.txt
└── README.md
```

GUI Commands for Testing:

```
# Install dependencies
pip install -r requirements.txt

# Run your visual GUI system
python src/main.py
```

Clearly document GUI interactions and provide illustrative screenshots in your `README.md`.

Submission Guidelines:

- Create a clearly structured **public GitHub repository**.
- Include comprehensive setup instructions and visual examples in your `README.md`.
- Submit the GitHub repository link through the provided email by **Sunday, 30th March, 11:59 PM**.

Evaluation Criteria:

- **Functionality & Correctness:** Complete implementation meeting all visual and functional requirements.
- **Code Quality:** Modular, readable, maintainable, and well-commented.
- **Visual Interface Quality:** Intuitive, user-friendly interface with clear interactions.
- **Traffic Negotiation Efficiency:** Effective collision avoidance, real-time rerouting, and lane management.
- **Dynamic Interactivity:** Smooth runtime robot and task management.
- **Robustness:** Thorough error handling, edge case handling, informative visual feedback.
- **Creative Enhancements:** Bonus points for additional innovative features or optimizations.

Nav Graph:

Sample 1: https://gist.github.com/naveenrobo/fbe659f375f5fb6aefb3bfd10a1cdbd7
Sample 2: https://gist.github.com/naveenrobo/71c0fb1913054eda0961f5072e47769b
Sample 3: https://gist.github.com/naveenrobo/0ca9f31f9748eea0f795c8b46e6d140a

Good Luck!