Traffic Agent Simulation 🚦 🚗

An agent-based traffic simulation built using the `autogen` framework, for visualizing the interaction between vehicles, traffic lights, pedestrian crossings, and parking areas. The simulation is organized around a **message-passing architecture**, where each element (vehicle, traffic light, pedestrian crossing, parking area) runs as an independent agent with its own logic.

Contribution of Group Members:

- Daniel Molina Pinel (50%): Focused on agents and worked on environment
- Mehmet Batuhan Duman (50%): Focused on environment and worked on agents.

Table of Contents

Features

- **Agent-Based Architecture:** Built using the `autogen` framework, each simulation element (vehicle, light, crossing, parking) is an autonomous agent communicating via asynchronous messages.
- **Asynchronous Simulation:** All agents operate and update their states concurrently.
- **Interactive Visualization:** Real-time Tkinter GUI (`vis/simui.py`) displays the map, agents, and their states (colors, text labels). Supports panning and zooming.
- **Configurable Map & Parameters:**
  - Define map layout (roads, intersections, agent placement) via JSON files (`map_config.json` , `basic_map_config.json`).
  - Override simulation parameters (timings, capacities, RL settings) via command-line arguments (see Simulation Input Parameters).
- **Road Network:**
  - Supports configurable road segments with capacity limits, one-way streets, and explicit connections.
  - Includes vehicle spawn and despawn points defined in the map configuration.
- **Vehicle Behavior:**
  - Vehicles navigate roads based on connections, attempt turns at intersections, and follow routes.
  - Respect traffic signals (lights, crossings) and road capacity, waiting when necessary.
  - Basic collision avoidance logic (primarily by checking obstacles ahead and road capacity).
- **Traffic Lights:**
  - Standard mode: Lights operate in coordinated groups (N/S vs E/W) on fixed timers.
  - RL mode (`--use-rl`): Optional `TrafficLightRLAssistant` agents use Q-learning (`rl/traffic_lihgt.py`) to dynamically adjust signals based on simulated queue lengths.
- **Pedestrian Crossings:**
  - Standard mode: Simulate pedestrian arrivals and occupy the crossing for a fixed or random duration.
  - RL mode (`--use-rl`): Optional `PedestrianCrossingRLAssistant` agents use Q-learning (`rl/pedestrian.py`) to decide when to allow crossings based on pedestrian queue length and road type.
- **Parking System (`complete` mode only):**
  - Supports different parking area types ("street", "roadside", "building") with configurable capacity, parking time, and exit time.
  - Vehicles can find nearby parking, request entry, park for a duration, and exit.
  - Standard mode: Parking areas manage occupancy and timed exits.
  - RL mode (`--use-rl`): Optional `ParkingRLAssistant` agents use Q-learning (`rl/parking.py`) to learn optimal exit strategies based on occupancy and duration.
- **Simulation Modes:** Easily switch between `basic` (no parking) and `complete` (with parking) modes via command-line argument.
- **Logging:** Key simulation events, agent actions, and final statistics are logged to a timestamped file in the `LOGS/` directory.

Getting Started

1. Prerequisites

- Python 3.7+ (Developed and tested with Python 3.11)

2. Setup Virtual Environment (Recommended)

It's recommended to use a virtual environment to manage dependencies.

```
# Navigate to the project directory
cd path/to/traffic_agents

# Create a virtual environment (if 'myenv' doesn't exist)
python -m venv myenv # Uncomment if needed

# Activate the virtual environment
# On Windows
.\myenv\Scripts\activate
# On macOS/Linux
source myenv/bin/activate
```

3. Install Dependencies

Make sure your virtual environment is activated, then install the required libraries:

```
pip install -r requirements.txt
```

4. Run the Simulation

There are two primary modes:

Basic Traffic Mode (without parking)

```
python main.py basic [OPTIONS]
```

Example: `python main.py basic --sim-time 60`

This launches a simplified simulation without parking areas, focusing only on vehicles, traffic lights, and pedestrian crossings. It reads `basic_map_config.json` for the map layout.

Complete Mode (with parking system)

```
python main.py complete [OPTIONS]
# Or simply (defaults to complete mode):
python main.py [OPTIONS]
```

Example: `python main.py --sim-time 120 --use-rl`

This launches the full simulation with all features, including parking and Reinforcement Learning. It reads `map_config.json` by default. A Tkinter window will appear, showing the simulation in real-time.

See Simulation Input Parameters for available `[OPTIONS]`.

Simulation Modes

1. **Basic Mode (`basic`)**
   - Uses `basic_map_config.json`
   - Focuses on traffic flow, traffic lights, and pedestrian crossings
   - Vehicles navigate intersections, avoid collisions, and obey signals
   - Ideal for studying fundamental traffic movement without parking
2. **Complete Mode (`complete`)**
   - Uses `map_config.json` (default if no mode is specified)
   - Includes **all** basic features plus parking areas
   - Vehicles may park, exit parking after some time, or skip parking if full
   - Parking areas have capacities, parking/exit times, and occupancy indicators
   - Shows the synergy between road traffic flow and parking availability

---

Configuration & Map Editing

The simulation reads from JSON configuration files (`map_config.json` or `basic_map_config.json`) to define:

- **Vehicles**: Initial state and properties.
- **Traffic Lights**: Location and type (standard/RL).
- **Pedestrian Crossings**: Location and type (standard/RL).
- **Parking Areas**: Location, capacity, type, timings.
- **Roads**: Geometry, connections, capacity, and other properties.

A typical `map_config.json` structure:

```
{
  "vehicles": [
    { "id": "vehicle_1", "x": 100, "y": 400, "spawn": true }
  ],
  "traffic_lights": [
    { "id": "traffic_light_1", "x": 100, "y": 0 }
  ],
  "crossings": [
    { "id": "crossing_1", "x": 100, "y": 100 }
  ],
  "parking_areas": [
    {
      "id": "street_parking_1",
      "x": 150,
      "y": 50,
```

```json
      "capacity": 3,
      "parking_time": 2,
      "exit_time": 1,
      "type": "street"
    }
  ],
  "roads": [
    {
      "id": "road_0",
      "x1": 0,
      "y1": 100,
      "x2": 700,
      "y2": 100,
      "capacity": 2,
      "one_way": false,
      "is_spawn_point": true,
      "is_despawn_point": false
    }
  ]
}
```

Map Elements

1. **Vehicles**
   - Define starting locations (`x` , `y`) or use `spawn: true` to use defined `spawn_points` in the map config.
2. **Traffic Lights**
   - Control traffic flow at intersections (either timed or RL-based).
3. **Pedestrian Crossings**
   - Occupy roads when pedestrians are crossing; vehicles must wait. Can be standard or RL-based.
4. **Parking Areas**
   - Define `capacity` , `parking_time` , `exit_time` , and `type` ("street", "roadside", or "building").
5. **Roads**
   - Each road has start/end coordinates (`x1, y1, x2, y2`), a unique `id` , and optional properties like `capacity` , `one_way` (boolean), `is_spawn_point` (boolean), `is_despawn_point` (boolean), and `connections` (list of road IDs this road leads to).

Parking System

Overview

The simulation provides a flexible parking system where vehicles can decide to park if they find an available spot:

- **Street Parking**: Smaller capacity, faster parking/exit times.
- **Parking Buildings**: Larger capacity, potentially slower times for parking/exit.

Parking States in Vehicles

1. **Driving** – Default state while on the road.
2. **Parking** – Currently transitioning into a parking area (takes `parking_time` seconds).
3. **Parked** – Vehicle is stationary in the lot.
4. **Exiting** – Transitioning out of the parking area (takes `exit_time` seconds).
5. **Searching** – Checking if a parking area has capacity or deciding whether to park.

The parking areas visually change color based on occupancy (blue/orange/red) and display `(current occupancy / capacity)` to indicate how many vehicles are parked.

Implementation Details

Agent Architecture

- All simulation elements extend a base class `MyAssistant` (in `base.py`).
- Agents handle messages asynchronously with the `@message_handler` decorator.
- **Example**: A `ParkingAssistant` might receive a "park_vehicle" message from a vehicle and update its occupancy.

Collision Avoidance & Capacity

- Vehicles track their progress along roads, and can wait if another vehicle is too close or if a traffic light is red.
- Lane capacity is respected, so if the capacity is reached, incoming vehicles may slow or queue.

Visualization

- Built with Tkinter (`simui.py`).
- Each agent (vehicle, crossing, etc.) has a corresponding "visual object" in the UI, drawn on a canvas with shapes, colors, and text labels.
- The UI also includes side panels displaying agent info (like vehicle status, parking occupancy, etc.).

Understanding Vehicle Colors and Percentages

- **Vehicle Color**: Indicates the vehicle's current state:
  - **Green**: Parked.
  - **Blue**: Parking or Exiting parking.
  - **Orange**: Has experienced wait times (Orange = longer waits).
  - **Default Blue**: Driving normally, no significant waits.
  - **Gray**: Default/fallback color.
- **Percentage**: Shown next to the vehicle ID (`Vehicle1 (75%)`), this indicates the vehicle's progress along its current road segment (0% = start, 100% = end). Only shown when driving.

File-by-File Overview

1. `base.py`

- Defines `MyAssistant`, an abstract base agent class.
- Implements `handle_my_message_type` as a default message handler.
- Provides common placeholders for road property processing.

2. `parking.py`

- **ParkingAssistant** manages a parking area:
  - Tracks capacity, vehicles in parking, exit timers, etc.
  - Runs a background coroutine (`run_parking_area`) that updates parking states every second.
  - Responds to messages for parking requests, state queries, and exit notifications.
- **ParkingRLAssistant** uses reinforcement learning (`rl.parking.ParkingRL`) to decide when vehicles should exit, aiming to optimize space usage.

3. `pedestrian.py`

- Contains:
  1. **PedestrianCrossingAssistant** – Standard, rule-based pedestrian crossing. Random arrivals, queue simulation, occupancy toggles.
  2. **PedestrianCrossingRLAssistant** – Reinforcement learning variant that can learn optimal times to let pedestrians cross.

4. `traffic_light.py`

- Holds:
  1. **TrafficLightAssistant** – Standard agent. Groups traffic lights (e.g., `north_south`, `east_west`) and toggles them periodically based on a shared timer.
  2. **TrafficLightRLAssistant** – RL-based agent. Each light uses its own RL model to adjust signals based on simulated queue length or other feedback.

5. `rl/traffic_lihgt.py` (RL model)

- *(Note the typo in the filename)* Contains `TrafficlightRL`, the Q-learning model used by `TrafficLightRLAssistant`.
  - Actions: keep current light, switch to green, switch to red.
  - Reward function based on queue lengths and green light duration.

6. `rl/parking.py` (RL model)

- Contains `ParkingRL`, the Q-learning model used by `ParkingRLAssistant`.
  - Actions: stay parked, exit parking.
  - Reward function based on parking duration, occupancy, and capacity.

7. `rl/pedestrian.py` (RL model)

- Contains `PedestrianCrossingRL`, the Q-learning model used by `PedestrianCrossingRLAssistant`.
  - Actions: occupy crossing (stop traffic), free crossing (allow traffic).
  - Reward function based on pedestrian queue length and road type.

8. `vehicle.py`

- **VehicleAssistant** simulates vehicle movement and behavior:
  - Progresses along roads, can turn if roads intersect.
  - Checks for collisions or obstacles.
  - Manages parking states (`parking`, `parked`, `exiting`, etc.).
  - Despawns at road ends if configured.

9. `main.py`

- **Entry point** for the simulation:
  - Parses CLI arguments (e.g., `--sim-time`, `--use-rl`, `--parking-time`, etc.).
  - Loads `map_config.json` or `basic_map_config.json`.
  - Registers agents (vehicles, traffic lights, etc.) with the runtime.
  - Starts the Tkinter GUI and the main simulation loop (async).

10. `runtime.py`

- Sets up a `SingleThreadedAgentRuntime` instance from `autogen_core`.
- Registers a “root” assistant (`MyAssistant`) and returns the runtime to be used by `main.py`.

11. `simui.py`

- **GUI** code using Tkinter:
  - Renders the simulation elements on a zoomable/pannable canvas.
  - Includes scrollable info panels showing real-time agent details (status, occupancy, etc.).

Reinforcement Learning

Optional RL agents can be used for more dynamic control:

1. **ParkingRLAssistant** (using `rl.parking.ParkingRL`)
   - Learns when vehicles should exit parking spots based on occupancy, capacity, and duration parked.
2. **PedestrianCrossingRLAssistant** (using `rl.pedestrian.PedestrianCrossingRL`)
   - Learns when to allow pedestrians to cross based on queue length and road type.
3. **TrafficLightRLAssistant** (using `rl.traffic_lihgt.TrafficlightRL`)
   - Learns optimal signal timing based on simulated queue lengths.

Enable RL-based agents by adding `--use-rl` to the command line. This flag enables RL for *all* applicable agent types (Parking, Pedestrian Crossings, Traffic Lights). You can also tune RL parameters:

```
# Run complete mode with RL agents, epsilon=0.2, learning_rate=0.05
python main.py --use-rl --epsilon 0.2 --learning-rate 0.05
```

Simulation Input Parameters

You can override default simulation parameters using command-line arguments:

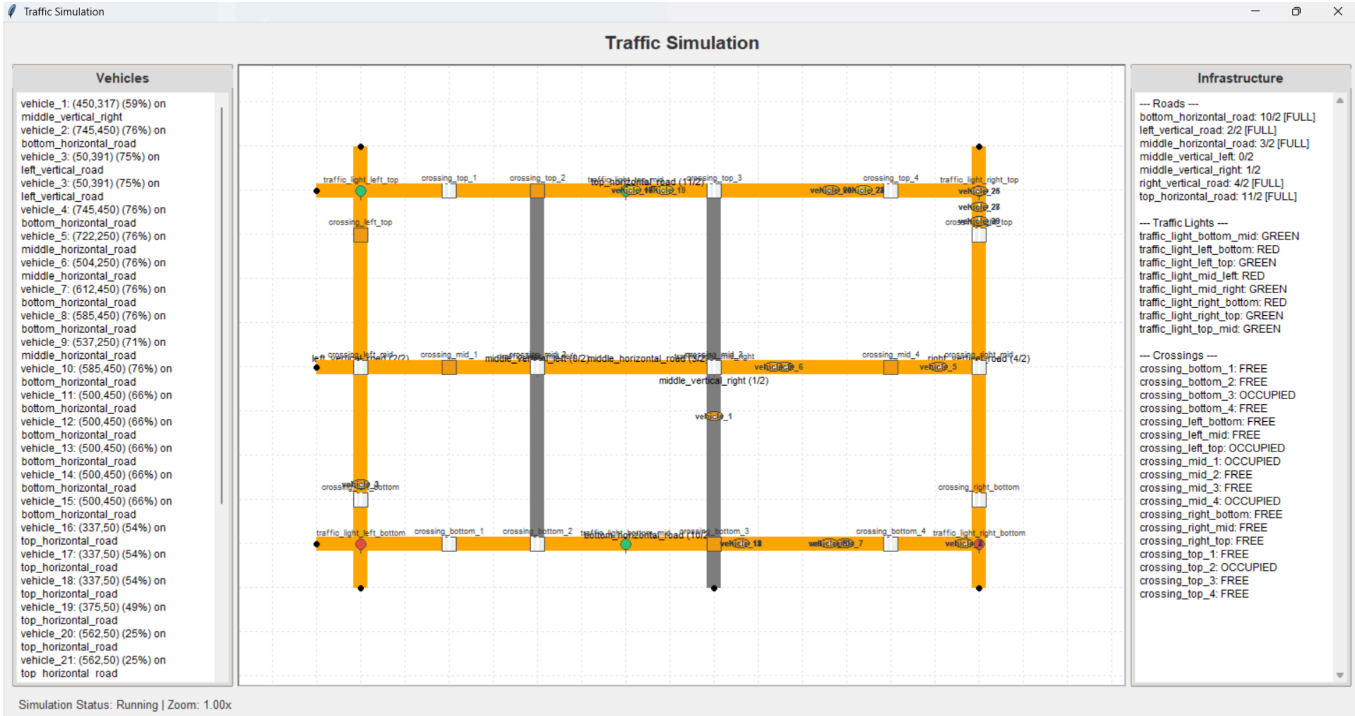
- `mode` : `basic` or `complete` (optional, defaults to `complete`). Placed *before* other options.
- `--sim-time INT` : Total simulation steps/seconds (default: 50).
- `--lane-capacity INT` : Default capacity for road segments (overrides config).
- `--traffic-light-wait INT` : Cycle time (seconds) for standard traffic lights.
- `--pedestrian-wait INT` : Crossing time (seconds) for standard pedestrian crossings.
- `--parking-time INT` : Average time (seconds) vehicles spend parking.
- `--exit-time INT` : Average time (seconds) vehicles spend exiting parking.
- `--parking-capacity INT` : Default capacity for parking areas (overrides config).
- `--use-rl` : Use RL agents instead of standard ones for traffic lights and crossings.
- `--epsilon FLOAT` : Exploration rate (epsilon) for RL agents (default: 0.1).
- `--learning-rate FLOAT` : Learning rate (alpha) for RL agents (default: 0.1).

Example Usage:

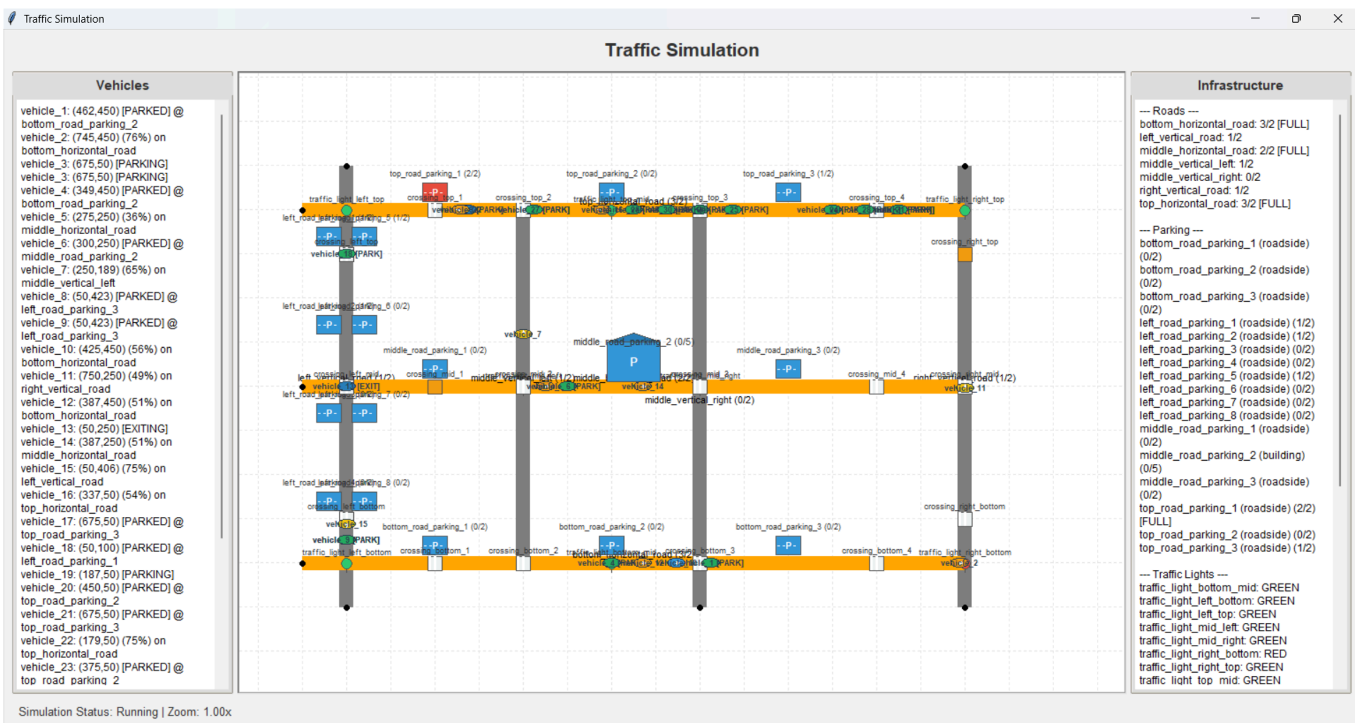
```
# Run basic mode for 100 steps with specific timings
python main.py basic --sim-time 100 --traffic-light-wait 6 --pedestrian-wait 4

# Run complete mode for 200 steps using RL agents with custom parameters
python main.py complete --sim-time 200 --use-rl --epsilon 0.15 --learning-rate 0.08 --
lane-capacity 3
```

Example Images from Simulation



The simulation in this picture is for the scenario without parking.



The simulation in this picture is for the scenario with parking.

Troubleshooting

1. **Tkinter Errors (`TclError`)**
   - Make sure your Python installation includes Tkinter.

- If you see errors like "display name and display number", ensure you are running in a graphical environment or have X11 forwarding configured if using SSH.

2. **Performance Issues / Lag**
   - A large number of vehicles or complex road networks can slow down the simulation and visualization.
   - Try reducing the `--sim-time` or simplifying the map configuration (`*.json` files).
   - The Tkinter visualization itself can be a bottleneck.
3. **Module Not Found Errors (e.g., `autogen_core` , `messages.types`)**
   - Ensure the virtual environment is activated (`source myenv/Scripts/activate` or in Windows environment `./myenv/Scripts/activate`).
   - Verify that `pip install -r requirements.txt` completed successfully.
   - If `autogen_core` or `messages.types` are custom/internal packages not in `requirements.txt` , make sure they are installed correctly in the active environment (e.g., using `pip install -e path/to/package` for local packages).
4. **Vehicles Not Moving / Parking / Turning Correctly**
   - Check the console output for error messages or warnings (e.g., "Blocked by...", "Parking full", "Agent not found", "Skipping turn...", "Invalid spawn point...").
   - Verify road `connections` in the JSON config. Incorrect or missing connections can prevent turns.
   - Ensure `spawn_points` reference valid `road_id` s.
   - Check if `capacity` limits on roads or parking areas are being hit.
5. **Configuration Not Applied**
   - Double-check the JSON syntax in `map_config.json` or `basic_map_config.json` .
   - Ensure you are running `python main.py` from the directory containing the JSON files and the `main.py` script.
   - Verify command-line arguments are spelled correctly and have the right types (e.g., `--sim-time 100` , not `--sim-time=100`).