Presented by Group Nine Yin Manaul

SDPAS

Smart Dynamic Priority Avoidance System

B1128015 Alan

B1128019 Jason

Adivisor : Dr. Bagnan

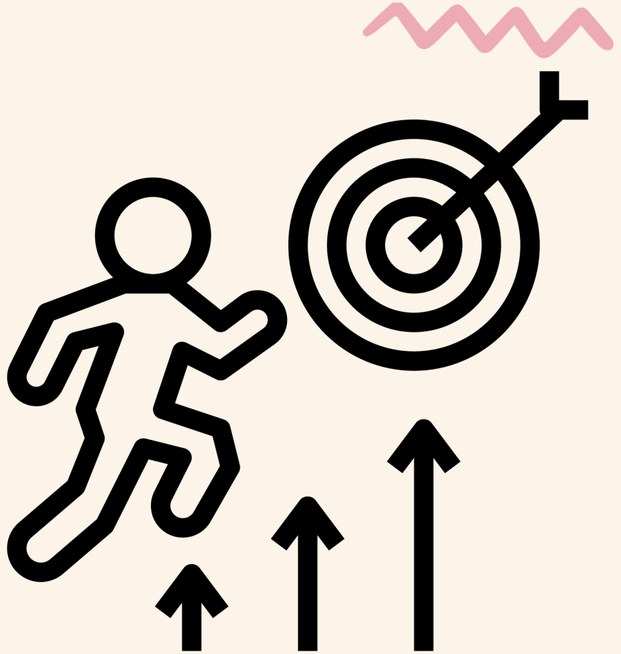
Date : 6/4

Problem

- Traffic Congestion
  - High vehicle density at intersections during peak hours causes long queues, increased travel time, and wasted fuel.
  - Fixed-time traffic signals cannot adapt to real-time fluctuations, leading to stop-and-go cycles and overall network inefficiency.
- Emergency Vehicle Delay
  - Traditional signal plans do not recognize approaching ambulances or fire trucks, forcing them to wait at red lights.
  - Delayed passage of emergency vehicles increases response time and may endanger lives.

Motivation

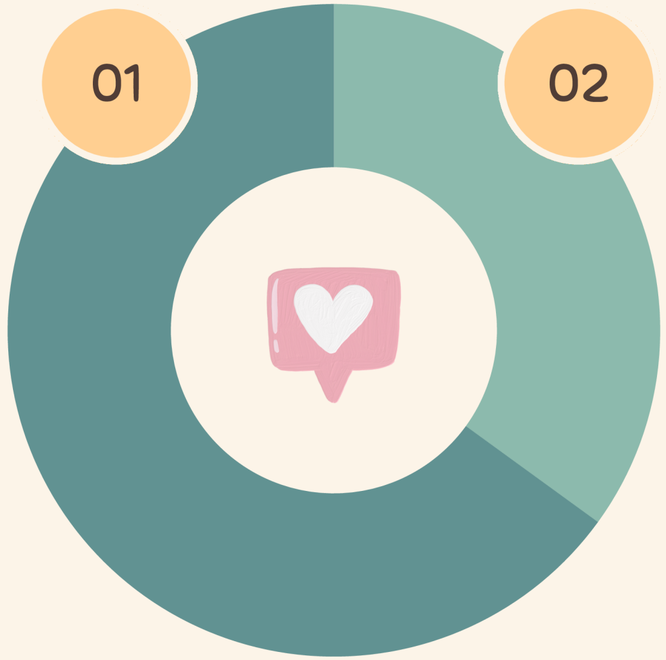
- Alleviate Congestion via Adaptive Control
  - Use Reinforcement Learning (RL) to tailor signal timings dynamically, minimizing overall waiting time and smoothing traffic flow.
  - A data-driven policy can react to sudden surges in vehicle count (e.g., after an event or during rush hour).
- Prioritize Emergency Vehicles Safely
  - Integrate emergency-vehicle detection (e.g., via TraCI subscription or external sensors) to temporarily override normal cycles and give red lights.
  - Ensure ambulances and fire trucks experience minimal delay, thereby reducing critical response times and improving public safety.



Objective

1 : Wait time.

- Reduce Queue Lengths
- Optimize Phase Duration
- Balance E–W and N–S Traffic
- Improve Intersection Throughput



2 : Emergency Priroty.

- Detect Approaching Emergency Vehicles
- Override Normal Signal Cycle
- Minimize Response Delay
- Maintain Safety for Other Vehicles

Environment

Environment

- SUMO Version & Configuration
- Network Topology

- SUMO (Simulation of Urban MObility) used as the core traffic simulator.
- Python-TraCI API
- Import OpenStreetMap location from the Taipei main trainstation

Environment

Environment

- Traffic Demand and Vehicle Types
- Reinforcement Learning Environment

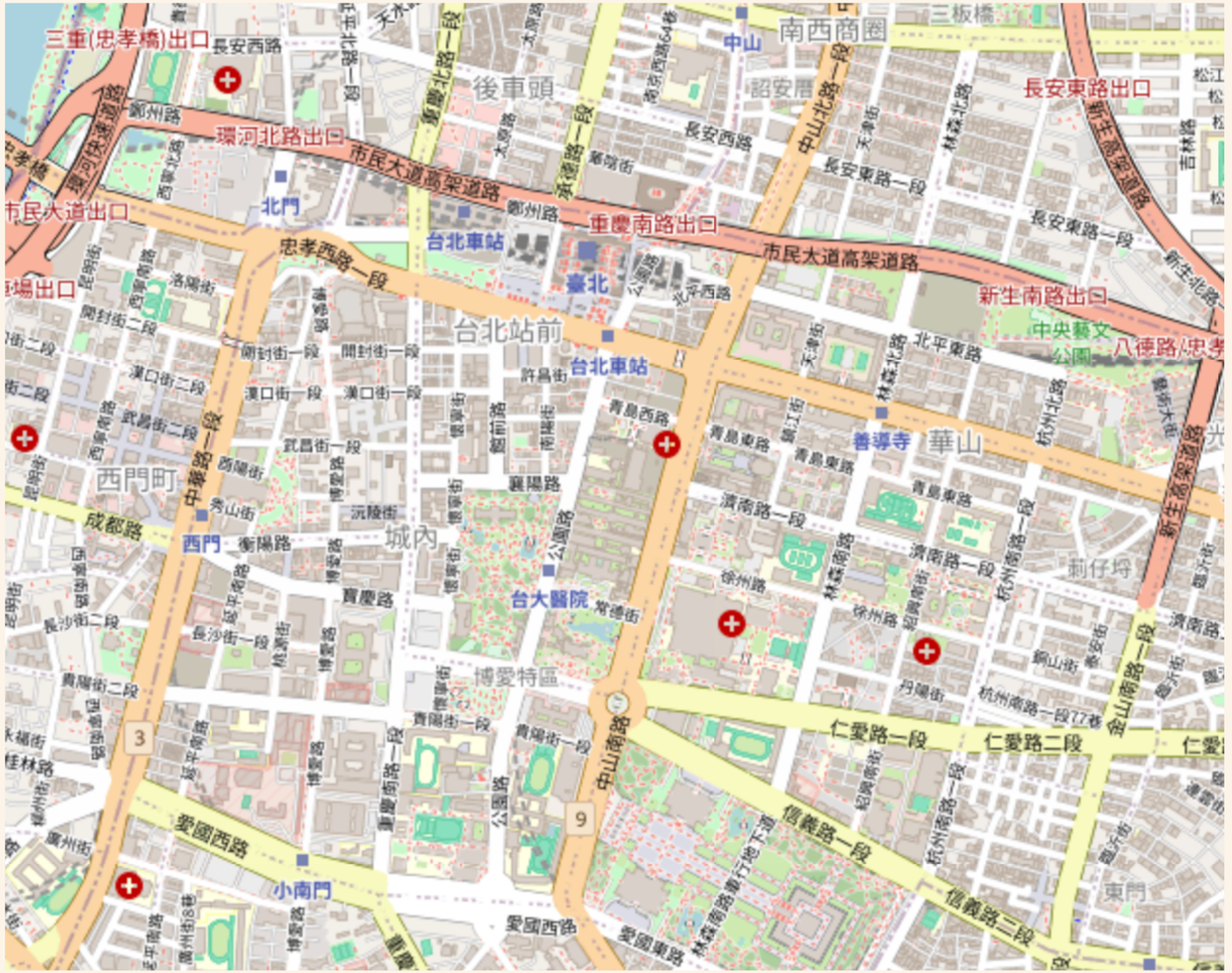
- Peak hours: 800–1,000 vehicles/hour per approach.
- Off-peak hours: 200–300 vehicles/hour per approach.
- Action space: two discrete actions (0 = grant East–West green, 1 = grant North–South green).
- Reward signal: negative aggregate waiting time difference with penalty for phase switching.

Dataset

Dataset

- Data Source and Preprocessing
- Version Control and Reproducibility

- Base road geometry imported from OSM (January 2025).
- All SUMO configuration files (.sumocfg, .net.xml, .rou.xml)


三重(忠孝橋)出口
長安西路
環河北路出口
市民大道出口
北門
台北車站
重慶南路出口
市民大道高架道路
忠孝西路一段
臺北
台北站前
台北車站
中山
南西商圈
後車頭
長安東路出口
新生南路出口
中央藝文公園
八德路/忠孝
善導寺
華山
西門町
西門
城內
台大醫院
博愛特區
小南門
愛國西路
仁愛路一段
仁愛路二段
信義路一段
信義路二段
東門
剎仔埒

System Architecture

Contents

01 Traffic Environment & SUMO Integration

02 Deep Q-Network & Replay Buffer

03 Training Workflow

04 Supporting Modules & Utilities

Traffic Environment SUMO Integration

- SUMO + TraCI Interface
  - The TrafficEnv class is responsible for launching the SUMO simulation (traci.start() / traci.load()) and subscribing to each lane's vehicle counts.
  - In __init__, it loads the list of traffic light IDs, selects an appropriate TL_ID; in reset(), it resets the environment; in step(), it executes a number of simulation steps (traci.simulationStep()) based on the action provided by the agent.
- Intersection Configuration & Lane Classification
  - Dynamically retrieves all lanes controlled by TL_ID at the intersection, then uses traci.lane.getAngle() to distinguish North–South lanes from East–West lanes.
  - Listens for LAST_STEP_VEHICLE_NUMBER on each lane, using this as one of the observation features.

Deep Q-Network Replay Buffer

- QNetwork (Neural Network Definition)
  - Three fully connected layers:
    - Linear(input_dim, hidden_dim) → ReLU
    - Linear(hidden_dim, hidden_dim) → ReLU
    - Linear(hidden_dim, output_dim)
  - Input dimension is fixed at 4 (NS vehicle count, EW vehicle count, current signal phase, elapsed phase time); output dimension is 2 (0 = EW green, 1 = NS green).
- ReplayBuffer (Experience Replay Pool)
  - Uses collections.deque(maxlen=BUFFER_CAPACITY) to store transition samples (s, a, r, s_next, done).
  - push() adds new samples to the buffer; sample(batch_size) randomly samples a batch and returns tensor versions for DQN training.

Training Workflow 1

- run_episode (Single-Episode Training Process)
  - For each episode, it creates a new TrafficEnv instance and calls reset() to obtain the initial state.
  - Uses an ε-greedy policy with policy_net to select an action, calls env.step(action) to receive (next_state, reward, done), then pushes the tuple into the ReplayBuffer.
  - When the buffer size ≥ BATCH_SIZE, it uses mixed precision (amp.autocast() / amp.GradScaler()) to perform a DQN update and, if necessary, synchronize target_net.
  - At the end of the episode, closes the environment (env.close()) and returns total reward, average loss, average waiting time, and average Q-value statistics.

Training Workflow II

- main (Overall Training Loop)
  - Initializes all settings: loads CFG hyperparameters, checks if SUMO_CONFIG exists, creates log directories, and initializes policy_net / target_net / optimizer / buffer, etc.
  - Uses TrafficEnv once to collect static network information (number of edges, number of lanes, total lane length), then enters for ep in range(NUM_EPISODES):
    - Writes the results from run_episode into the metrics CSV and temporarily stores them in memory.
    - Writes to disk every 20 episodes to avoid excessive I/O.
    - If the current episode's ep_reward exceeds the best reward so far, saves a checkpoint.
    - Logs the average reward over the last 100 episodes every 100 episodes.
- After training completes, saves both policy_best and policy_final models separately and generates five diagnostic plots (loss, waiting time, Q-value, ε, reward).

Supporting Modules
Utilities 1

- Reward Calculation & Emergency Vehicle Handling
  - reward_delta(prev_wait, curr_wait, prev_phase, curr_phase): Computes the waiting-time difference as a “smooth reward” and applies a PENALTY_SWITCH penalty if the phase switches.
  - get_ambulance_edges(route_files): Parses all .rou.xml files to collect edge IDs associated with emergency vehicle routes, used to identify which traffic lights are relevant for emergency vehicles.
  - set_full_red(tls_id, duration) / clear_full_red(tls_id): Forces an all-red signal for a specified duration to support emergency vehicle priority.
  - handle_emergency_blockage(veh_id, wait_thr, occupy_time): If an emergency vehicle has been waiting longer than wait_thr, attempts to change its lane or re-route it to avoid being stuck.

Supporting Modules
Utilities II

- Route File Sanitization (Automatically Filtering Invalid Routes)
  - _get_net_path(sumocfg_path): Reads the corresponding .net.xml network file path from a given .sumocfg.
  - sanitize_route(net_path, route_path): If a .rou.xml file contains routes referencing edges not present in the network, removes affected <vehicle> or <route> elements and outputs a temporary sanitized .rou.xml.
- Logging & Plotting
  - CSV Writer: In main(), opens metrics.csv and collects [episode, total_reward, mean_loss, mean_wait, mean_q, epsilon, num_edges, num_lanes, total_lane_length] each episode.
  - Matplotlib Plots: After training, automatically generates five diagnostic plots and saves them under training_logs/<run_id>/.
  - suppress_output(): Used inside run_episode to temporarily redirect stdout/stderr to /dev/null, preventing large amounts of SUMO output from cluttering the terminal.

Algorithm – DQN / SARSA

DQN

- Network Structure
  - QNetwork(input_dim=4, hidden_dim=256, output_dim=2)
  - Inputs: [NS count, EW count, current phase, elapsed phase time]
  - Outputs: Q-values for {EW-green, NS-green}.
- Experience Replay
  - ReplayBuffer holds (s, a, r, s_next, done) in a deque.
  - Sample minibatch when buffer size ≥ BATCH_SIZE.
- Update Rule
  - Compute q_preds = policy_net(s_b).gather(...).
  - Compute target with q_next = target_net(s2_b).max(1)[0].
  - Loss = MSE(q_preds, r_b + γ·q_next·(1−done)).
  - Sync target_net from policy_net every TARGET_UPDATE steps.

Algorithm – DQN / SARSA

SARSA

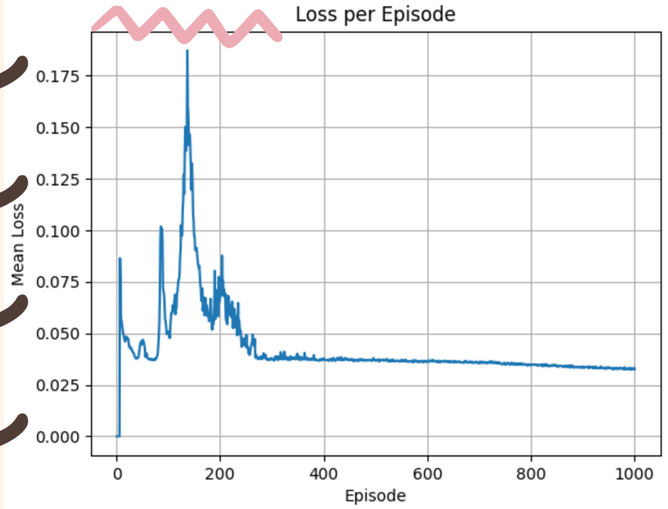
- SARSA (On-Policy Variant)
  - Target Calculation
  - Uses the same network for next-state Q:
    - next_q_all = policy_net(s2_b)
    - a_next = next_q_all.argmax(1)
    - q_next = next_q_all.gather(1, a_next.unsqueeze(1)).squeeze(1)
  - Target = r_b + γ·q_next·(1−done).
- Key Difference
  - DQN: off-policy target from target_net.max(…).
  - SARSA: on-policy target using policy_net's own greedy action.
- Code Integration
  - In run_episode(), branch on CFG.USE_SARSA to select which q_next formula to use; otherwise, all training steps remain the same.

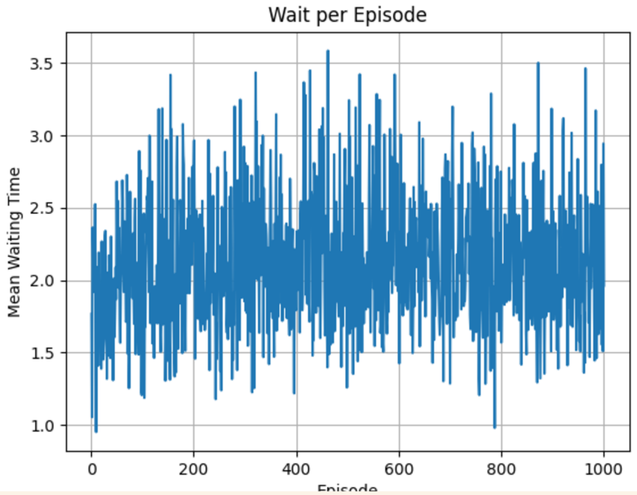
Global Hyperparameter

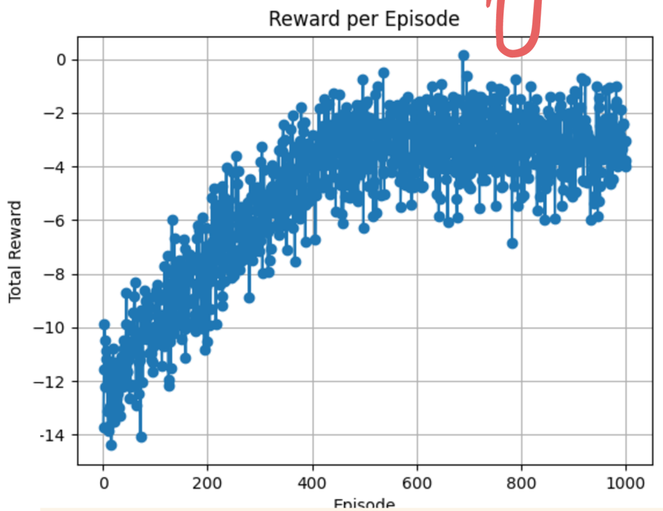
Class CFG

- Reproducibility
  - SEED = 42
- Algorithm Selection
  - USE_SARSA = True / False
- Replay & Training
  - BUFFER_CAPACITY = 100,000, BATCH_SIZE = 1024, GAMMA = 0.98
- Exploration Schedule
  - EPS_INIT = 1.0, EPS_DECAY = 0.995, EPS_MIN = 0.1
- Phase Switching Constraint
  - MIN_GREEN_STEPS = 5

Training Curves







01

Quantitative Results

Live Demo (Smart Screen)

Contents

Video Link

CONCLUSION & FUTURE WORK

- Conclusion
  - Effective Congestion Reduction
  - Emergency Vehicle Priority
  - Scalability & Reproducibility

- Future Work
  - Multi-Intersection Coordination
  - Pedestrian & Bicycle Phases
  - Dynamic Demand & Real-World Validation
  - Adaptive Reward Engineering

Team & Contact / Q&A

Thank You

By Group Borcelle