Final Project Report for CS 175

Project Title: Traffic Control Systems Using Reinforcement Learning
Project Number: 12

Student Names
Rohan Mistry, 23895536, rohandm@uci.edu
Derek Xu, 31628459, dkxu@uci.edu
Harry Leung, 29279838, hgleung@uci.edu

1. Introduction and Problem Statement

This project aims to develop a Traffic Control System using Reinforcement Learning (RL) to optimize urban traffic flow by making real-time traffic light decisions. Our approach involves training an agent that controls the color and duration of traffic lights using Multi-Agent Proximal Policy Optimization (MAPPO) to coordinate with neighboring agents and optimize traffic congestion. The model will be implemented and trained in the Gymnasium-integrated environments CityFlow to learn effective traffic signal timings over time.

Traditional traffic signal control methods typically rely on fixed time intervals or simple rule-based systems, which often fail to adapt to varying traffic patterns. The goal of this project is to develop a system that dynamically adjusts traffic light timings in real time based on the traffic conditions at different intersections, reducing congestion and improving overall traffic flow. The input to the system will be real-time traffic data, such as vehicle counts, and traffic speeds, which are typically collected from sensors at different intersections. The output will be optimal traffic light timings that adjust based on the traffic data received.

2. Related Work

Various studies have attempted to apply Machine Learning (ML) techniques such as Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Graph Neural Networks (GNN) to the problem of traffic control in the past. For instance, a study titled *Coordinated Deep Reinforcement Learners for Traffic Light Control* by the University of Amsterdam used Deep Q-Learning (DQL) to try to solve this problem. Specifically, each intersection is defined as an agent trying to learn an optimal policy based on traffic conditions, but this approach faces stability issues in dynamic environments where traffic conditions change frequently. Another study titled *Developing adaptive traffic signal control by actor–critic and direct exploration methods* uses an Actor-Critic (AC) Algorithm, where a critic provides feedback to an actor to help it improve, which is useful in environments where optimal actions vary frequently. However, this approach is prone to instability along with convergence issues in highly dynamic environments. Due to various issues faced by previous traffic control solutions using ML, especially with regards to stability, it is necessary to develop new methods of ML-based traffic control, which is what we aim to do with MAPPO, an RL method that prioritizes stability during training.

3. Data Sets

Rather than fit our model and environment to existing traffic datasets, we decided to use simulated data generated from CityFlow so that we could define the parameters based on how we wanted to configure the project. Through this process, we were able to generate several customized traffic grid scenarios of varying sizes and layouts, each with predefined road-intersection networks, traffic flow, and vehicle information. The data is stored within a road network JSON file [see Figure 2 and Figure 3] and a traffic flow JSON file.

The road network file defines the road network structure. CityFlow's road network implementation primarily consists of *intersections* and *roads* (can be thought of as nodes and edges of a graph).

- *Road* represents a directional road from one *intersection* to another *intersection* with road-specific properties
    - Each *road* has a defined start location and end location as well as a set of intersections including the one it starts from, the one it ends at, and the others it goes through
    - A *road* may contain multiple lanes, each with defined widths and speed limits
- *Intersection* is where roads intersect
    - Each *intersection* has a defined location, width, set of roads that convene towards it, several *roadlinks*, and light phases which define the different possible light signal configurations of that intersection
    - Each *roadlink* connects two roads of the intersection and can be controlled by traffic signals
- A *roadlink* may contain several *lanelinks*
    - Each *lanelink* represents a specific path from one lane of the incoming road to one lane of the outgoing road

Each flow in the traffic flow file consists of the following fields:

- `vehicle`: defines the parameter of the vehicle
    - `length`: length of the vehicle
    - `width`: width of the vehicle
    - `maxPosAcc`: maximum acceleration (in m/s)
    - `maxNegAcc`: maximum deceleration (in m/s)
    - `usualPosAcc`: usual acceleration (in m/s)
    - `usualNegAcc`: usual deceleration (in m/s)
    - `minGap`: minimum acceptable gap with leading vehicle (in meters)
    - `maxSpeed`: maximum cruising speed (in m/s)
    - `headwayTime`: desired headway time (in seconds) with leading vehicle to keep current `speed * headwayTime` gap
- `route`: defines the route
    - All vehicles of this flow follow this route
    - Specify source and destination, optionally some anchor points and router will connect them with shortest paths automatically
- `interval`: defines the interval of consecutive vehicles (in seconds)
    - If interval is too small, vehicles may not be able to enter the road due to blockage, will be held and let go once there is enough space
- `startTime`, `endTime`: simulation will generate vehicles between time `[startTime, endTime]` (in seconds, inclusive)

For more information about how this data is utilized in our environment, see the "Software" section.

For validation of our model, we compared the results of our training against fixed traffic signal data, in which the traffic signals have a predefined set of rules that they follow for switching signal lights and do not adapt based on new data. We planned on using publicly available traffic datasets, specifically the "Traffic Flow Forecasting" dataset from the UC Irvine Machine Learning Repository, which allows for the creation of a model predicting traffic flow data from various details from various sensors in urban areas. This dataset contains traffic counts, vehicle speeds, and other external conditions. However, we could not find a pre-existing dataset that included all the features we planned on tracking and was configured in such a way that was integrable with CityFlow so we instead opted for simulated training data. We still plan on using the "Traffic Flow Forecasting" dataset for additional validation of our trained model but will not use it to train our model.

4. Technical Approach

To train our model, we used multi-agent proximal policy optimization (MAPPO), a training method based on PPO, a related technique that limits the size of policy updates to improve agent stability. This is done so that the training has a higher chance of converging to an optimal solution, which is more likely with smaller updates. However, policy updates cannot be too small, otherwise the training would become too slow. Therefore, PPO uses a clipped surrogate objective function that confines the size of the policy update using a clip. The version of PPO used in our approach comes from Ray's RLlib. Because each traffic light at each intersection is controlled by an individual agent, we opted for MAPPO rather than a traditional single-agent PPO algorithm. MAPPO is suited for training in situations with large action spaces, and in this case, it allows different agents to learn collaboratively as well.

We also implemented a custom environment to use during training. Because CityFlow is itself a basic environment template for this problem and does not have all the necessary methods and configurations for our PPO implementations, we created our environment as a wrapper around the CityFlow API. The observation space for each agent (local states) includes intersection-specific data such as total vehicle count, waiting vehicle count, and throughput. The traffic states from all intersections are then combined to form the global state of the entire grid. The action space for each agent represents the possible traffic signal configurations at that intersection (which lanes to turn the green light on for and which lanes to turn the red light on for). Implementing PPO for each agent to learn and adjust their traffic signals allows for handling large, complex action spaces and prevents catastrophic policy updates during training. Because the implementation is based on centralized training and decentralized execution, each intersection is simulated as an independent agent and the environment logic is centralized so that agents can share their local states during training as a global state but each acts independently during execution.

The reward function is based on traffic flow efficiency, penalizing delays and encouraging smooth transitions through intersections. Rewards are distributed to agents based on reduced queue lengths, reduced waiting, and maximized throughput.

We first tested our environment with random actions to ensure smooth state-action transitions and that our customized setup is compatible with CityFlow's API.

Our environment is inherited from Ray's `MultiAgentEnv` class and includes the necessary methods to be compatible with Ray's PPO implementation, such as `reset`, which resets the environment, and `step`, which takes an action and updates the environment accordingly. We imported Ray's RLlib library which is specifically designed for MAPPO training. The policy network consists of per-intersection local states as input and a probability distribution over actions as output. During the centralized training stage, the model learns using the global state (all agents' observations) and outputs a single value for the entire system, helping to coordinate actions across agents. During the decentralized execution stage, each agent has its own policy network that uses only local observations during execution. During training, we trained agents across hundreds of episodes and logged key metrics. After training steps, each agent uses its local policy network for decision-making. During execution, agents observe their local states and select actions based on their individual trained policies. The CityFlow API is used to send states during each step and receive actions during training.

After initial training, we focused on MAPPO-specific hyperparameters and used Ray's Tune tool to automate the tuning process. We experimented with learning rates and reward scaling among other factors. Traffic metrics were saved and analyzed and then we adjusted the reward function to the best of our capabilities to ensure it balances its coefficients and its penalties and bonuses steer the policy to better effectiveness.

[see Figure 5 for diagram of components of our technical approach]

5. Software

Python is the primary programming language used, both for our software and the software outsourced externally, to implement the model and environment. GitHub is used for version control and collaboration among group members on the codebase.

a. Software by us

i. Custom RL Environment for Traffic Signal Control

To facilitate development of our PPO model, we first developed a custom traffic signal control environment `TrafficSignalEnv` that works as a wrapper around the CityFlow API and Ray's RLlib `MultiAgentEnv`. To initialize the environment, three variables need to be passed in as arguments: `max_steps`, which is the maximum number of steps to iterate through for each training episode; `config_file`, which is the relative path to a JSON configuration file necessary for CityFlow to specify settings and configuration parameters during the simulation; and `road_network_file`, which is the relative path to the JSON file consisting of the components of the generated traffic network. Calling `cityflow.Engine(config_file, thread_num=1)` creates the CityFlow engine for the simulation of this episode.

The `_parse_network(road_network_file)` method reads the provided road network file and groups all of the intersections in the grid together and groups all of the roads in the grid together. The observation space is then defined as a Gymnasium dictionary space with observations assigned to each agent. The action space is also defined as a Gymnasium dictionary space with light phase configurations for each intersection to the associated agent. The `_get_observation_for_agent(agent_idx)` method retrieves the observations (vehicle counts, waiting counts, etc.) for the specified agent, padded as a NumPy array. The `_get_rewards()` method uses dynamic normalization to find the normalized values for vehicle count, waiting count, and calculated throughput, useful for scaling to larger grid sizes. These parameters are then assigned coefficients based on the tuning process which calculate an overall reward for each agent, collected together in a list and returned. The `_is_done()` method simply checks if the engine has reached `max_steps`. We first created a `fixed_signal_control(env)` method to test the integration of our environment with CityFlow's API and observe the results of our simulations with fixed signals as the baseline for our testing and evaluation steps.

For our PPO configuration, we created an initial training loop function, a hyperparameter tuning function, and an evaluation loop function. First, we defined the environment configuration for our training and registered a custom environment using our `TrafficSignalEnv` class. Then, we defined a dictionary `policies` to track the policies for each agent. In `train_PPO()`, we initialized the PPO configurations and applied them using Ray's predefined `PPOConfig` class. We used the PyTorch framework, defined environment runners, and inserted training hyperparameters such as:

- `gamma=0.99` (discount factor)
- `train_batch_size=2048` (total batch size for training)
- `minibatch_size=64` (minibatch size)
- `num_epochs=10` (number of epochs to iterate over each batch)
- `grad_clip = 0.5` (gradient clipping value)
- `clip_param = 0.2` (clipping parameter for PPO)

We also defined a policy mapping function to map policies to their respective agents. We then used Ray's `PPO` class which initialized a PPO algorithm instance and then executed the training loop, calling Ray's `train()` method and logging meaningful data in csv files for each training iteration. Some of the metrics we tracked included agent-specific metrics such as total loss, policy loss, value function optimization, entropy, and current learning rate; environment metrics such as average reward per episode and average length of episodes; and system metrics such as learning time, steps sampled, and overall throughput. Once the training loop was complete, we saved the trained policy so that we could pass it as a parameter to our `evaluate_policy(trained_policy)` function. This function runs a training iteration on a newly customized environment using the newly trained policy and tracking evaluation results.

Our final function is `tune_hyperparameters()` which uses Ray Tune's `Tuner` class to define various parameters for automated tuning. We passed in lists of potential values for each hyperparameter so that it could iterate through each configuration set and find the optimal set of hyperparameters. This process requires significant compute power, so we had to scale back the tuning to make it more feasible with our limited available resources.

b. Software by Others

i. CityFlow

CityFlow is a multi-agent RL environment designed for large-scale city traffic scenarios which consists of several tools that we used. We used the `Engine` feature to initialize our environment and the grid scenario generation feature [see Figure 1] to create our simulated datasets, consisting of customized traffic scenarios, predefined intersection layouts, traffic flow configurations, and detailed vehicle information. Each grid scenario consists of a road network JSON file and a traffic flow JSON file.

CityFlow has a data access API which includes methods such as:

- `get_vehicle_count()`: returns the number of total running vehicles
- `get_vehicles(include_waiting=False)`: returns a list of all vehicle IDs
    - Includes or excludes vehicles that are in a lane's waiting buffer based on value of `include_waiting`
- `get_lane_vehicle_count()`: returns a dictionary of running vehicles in a specified lane
- `get_lane_waiting_vehicle_count()`: returns a dictionary of waiting vehicles in a specified lane
- `get_vehicle_info(vehicle_id)`: returns a dictionary of various information of the given vehicle like speed and current road
- `get_vehicle_speed()`: returns the speed of each vehicle

CityFlow also has a control API which includes methods such as:

- `set_tl_phase(intersection_id, phase_id)`: sets the phase of the traffic light of the specified intersection to the specified phase

For each episode of the training, replay files are generated automatically and can be inserted into CityFlow's online visualization simulator. Using the simulator, the user can maneuver through each step of the episode and see exactly where all the vehicles are located within the traffic grid, what the current traffic light signal configurations are, and observe how successful the trained model is at optimizing the flow of the traffic. For more information about CityFlow, see https://cityflow.readthedocs.io/.

ii. Gymnasium

Gymnasium is an RL library primarily used for creating RL models and comparing their performance. Other than being useful for making RL models, Gymnasium also offers a standard API facilitating the creation of RL models as well. In our code, we used Gymnasium's spaces module to represent the observation and action spaces of our environment. For more information about Gymnasium, see https://gymnasium.farama.org/index.html.

iii. NumPy

Numpy is a Python library for processing arrays of numerical elements and performing operations with them. Common ways of processing arrays include performing memberwise operations on each of the elements of arrays with identical shapes. NumPy is primarily used in our code for representing states in the custom environment we made. For more information about Gymnasium, see https://numpy.org/.

iv. Ray

Ray is a framework for scalable ML that can be used for a variety of tasks, including MAPPO. Its RL capabilities via its RLlib library provide an abstract base class that can be used to create environments for RL and run the PPO algorithm. Ray provides valuable flexibility in its configuration for training the model on the PPO algorithm for many iterations and tuning various hyperparameters for optimized results. Ray also has a Tune tool that was used to automate the hyperparameter tuning process. For more information about Ray, see https://docs.ray.io/en/latest/ray-overview/index.html.

6. Experiments and Evaluation

To evaluate the effectiveness of our reinforcement learning-based traffic control system, we compared the simulation environment that used the MAPPO implementation with various other traffic light signal control systems, such as a rule-based implementation. The experiments were conducted in Cityflow, which allowed for flexible and scalable testing across a variety of intersection configurations and traffic conditions.

The simulation setup involved a grid network of intersections with varying traffic densities, ranging from low congestion to peak-hour traffic scenarios. Within the MAPPO simulation, each intersection was controlled by an independent RL agent. Agents operated with a discrete set of actions corresponding to traffic signal phases.

Key hyperparameters for training were tuned based on a combination of grid search and domain expertise. These included learning rate, discount factor, and the coefficients of the reward function components: waiting time, queue length, and throughput. Training episodes consisted of 1,000 time steps, with evaluation conducted after every 100 episodes to assess performance improvements.

To quantify the system's performance, we employed the following metrics:

- Average Waiting Time: The mean time vehicles spent idling at intersections.
- Queue Length: The average number of vehicles in line at traffic lights.
- Throughput: The total number of vehicles that successfully passed through the intersections during a given simulation period.
- Reward Score: A composite measure derived from the reward function, combining the three metrics into a single evaluation criterion.

For comparison, we implemented a rule-based baseline which used static signal timings. These baselines served to benchmark the performance of our RL approach. As the PPO training progressed, the average reward score returned by the environment steadily increased over the fixed-action implementation, showing an improvement over the baseline system [see Figure 4 and Figure 6].

7. Discussion and Conclusion

We gained many insights from this project, both from the valuable experience in implementing RL algorithms for a real-world problem and collaborating with others on such a comprehensive undertaking. Though our model is still in the early stages of development for a fully-implemented MAPPO application, preliminary results are still positive, indicating that even a simplified version of what could be an incredibly robust application is far superior to fixed signals. We learned how to utilize existing RL environments and new frameworks to facilitate our customized model and about the significant complexities that come with developing multi-agent solutions and scaling them to larger-sized implementations. The results agreed with our expectation in that it seems that MAPPO is a much-preferred algorithm to many, if not all, of the alternatives that have been used before because it theoretically adapts for scenarios of a larger size and can be applied more directly to real-world applications. The gradient-based updates and balancing of exploration and exploitation dynamically minimized delay and traffic jams fairly well. However, to fully stabilize, we will need to run several thousand training episodes which was not feasible for us at this moment. We did face several complications during this project as well, some of which we expected and others which were a surprise and learning experience for us.

The first major challenge that we faced during the project was environment setup. We searched for existing setups but could not find anything that supported all of the features and intricacies that we were looking for. So, we created a customized environment class wrapper around the CityFlow API and Ray's MultiAgentEnv which took extensive time due to the serious learning curve of the CityFlow software and debugging to ensure that our implementation was configured correctly and that we could move onto the training process. Because of the multi-agent setup with centralized training but individual policies for each agent, keeping track of features for each intersection became very complex as well. Implementing and optimizing the PPO algorithm with tuning to monitor training behavior on several iterations was also a challenge. Due to limited compute power and the time crunch of completing the project, we are still working on scaling the problem to much larger grid scenarios and creating a more finely-tuned reward function with an expanded set of features. The multi-agent setup and elaborate CityFlow traffic scenarios made it tougher to expand to a more robust implementation.

One significant adjustment during the initial environment setup was our decision to pivot from utilizing the predefined dataset from the UC Irvine Machine Learning Repository and instead using simulated scenarios via CityFlow which provided more detailed data necessary for training. We also tested a couple different PPO implementations, including that of code found online and that of the stable_baselines3 library before realizing that they did not handle multi-agent implementations very well. Thus, we switched to Ray's RLlib library which was much more effective in implementing MAPPO in a way that fit well with our environment configurations and the CityFlow API. Throughout the project, we learned about the importance of tool selection because not every framework fits for every use case and we had to adapt to multiple tools before settling on the right ones.

If we were in charge of a research lab, we would find the necessary resources and tools to construct a much more elaborate model and have the capability to run several thousand training episodes and compute the hyperparameter configurations with much more precision. Exploring more advanced feature tracking for each vehicle, including incorporating environmental goals such as minimizing fuel consumption and reducing CO_2 emissions and each intersection and implementing continuous action spaces, would allow for more fine-tuned timing for signal switching and computing the duration and intensity of each light and more granular control. We would also make the problem more complex but also more realistic by reward shaping via experimenting with different reward structures that prioritize not only congestion reduction but also pedestrian crossings or emergency vehicle passage. We would also look into hierarchical policies such that in more complex traffic networks, traffic can be coordinated across much larger regions which might help scalability in urban cities. Future research could also explore how connected vehicle technologies such as Vehicle-to-Infrastructure (V2I) communication enhance the RL model's decision making. Real-time updates from connected vehicles might provide richer, more actionable data. Finally, collaborating with municipalities to deploy small-scale pilot programs using real intersections would allow us to validate our approaches and refine them based on live data and partnering with experts in urban planning could offer fresh perspectives in the design and structure of our model.

8. Appendix

Figure 1: CityFlow’s online simulator

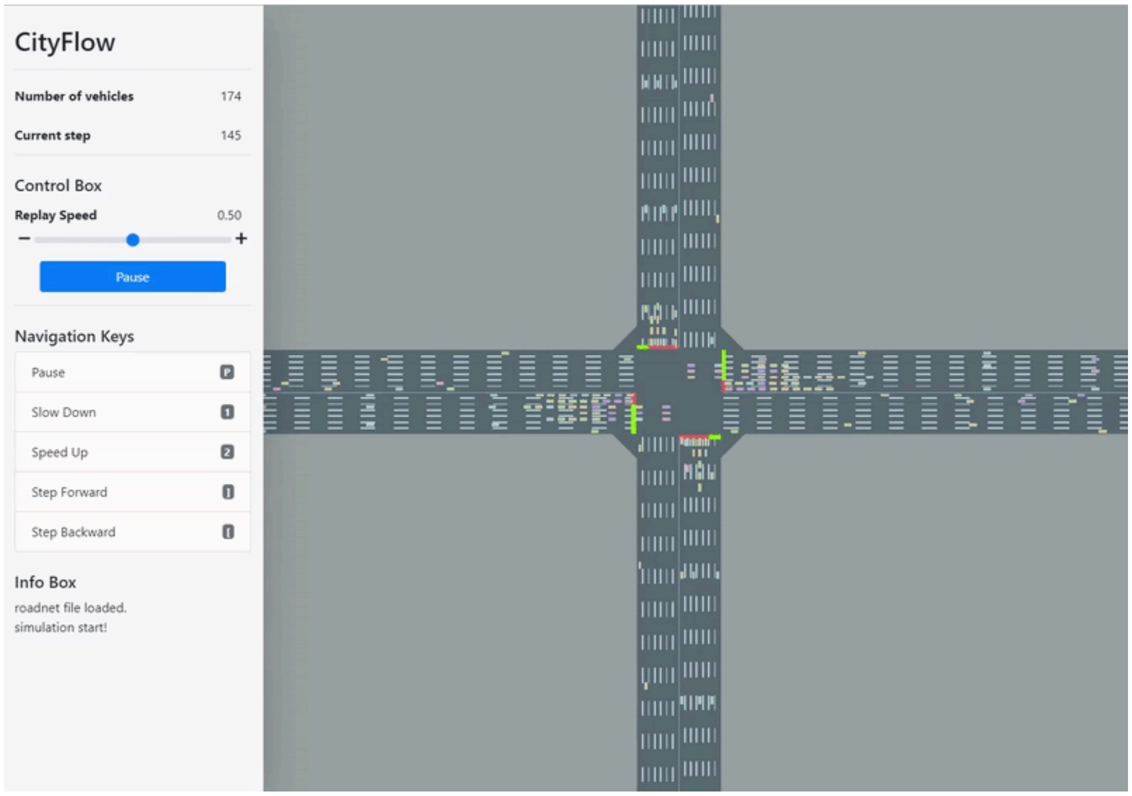


Figure 2: sample road network JSON file with explanations for each component

```
{
  "intersections": [
    {
      // id of the intersection
      "id": "intersection_1_0",
      // coordinate of center of intersection
      "point": {
        "x": 0,
        "y": 0
      },
      // width of the intersection
      "width": 10,
      // roads connected to the intersection
      "roads": [
        "road_1",
        "road_2"
      ],
      // roadLinks of the intersection
      "roadLinks": [
        {
          // 'turn_left', 'turn_right', 'go_straight'
          "type": "go_straight",
          // id of starting road
          "startRoad": "road_1",
          // id of ending road
          "endRoad": "road_2",
          // lanelinks of roadlink
          "laneLinks": [
            {
              // from startRoad's startLaneIndex lane to endRoad's endLaneIndex lane
              "startLaneIndex": 0,
              "endLaneIndex": 1,
              // points along the laneLink which describe the shape of laneLink
              "points": [
                {
                  "x": -10,
                  "y": 2
                },
                {
                  "x": 10,
                  "y": -2
                }
              ]
            }
          ]
        }
      ],
      // traffic light plan of the intersection
      "trafficLight": {
        "lightphases": [
          {
            // default duration of the phase
            "time": 30,
            // available roadLinks of current phase, index is the no. of roadlinks
defined above.
            "availableRoadLinks": [
              0,
              2
            ]
          }
        ]
      },
      // true if it's a peripheral intersection (if it only connects to one road)
      "virtual": false
    }
  ],
  "roads": [
    {
      // id of road
      "id": "road_1",
      // id of start intersection
      "startIntersection": "intersection_1",
      // id of end intersection
      "endIntersection": "intersection_2",
      // points along the road which describe the shape of the road
      "points": [
        {
          "x": -200,
          "y": 0
        },
        {
          "x": 0,
          "y": 0
        }
      ],
      // property of each lane
      "lanes": [
        {
          "width": 4,
          "maxSpeed": 16.67
        }
      ]
    }
  ]
}
```

Figure 3: illustration of basic 1x2 grid road network

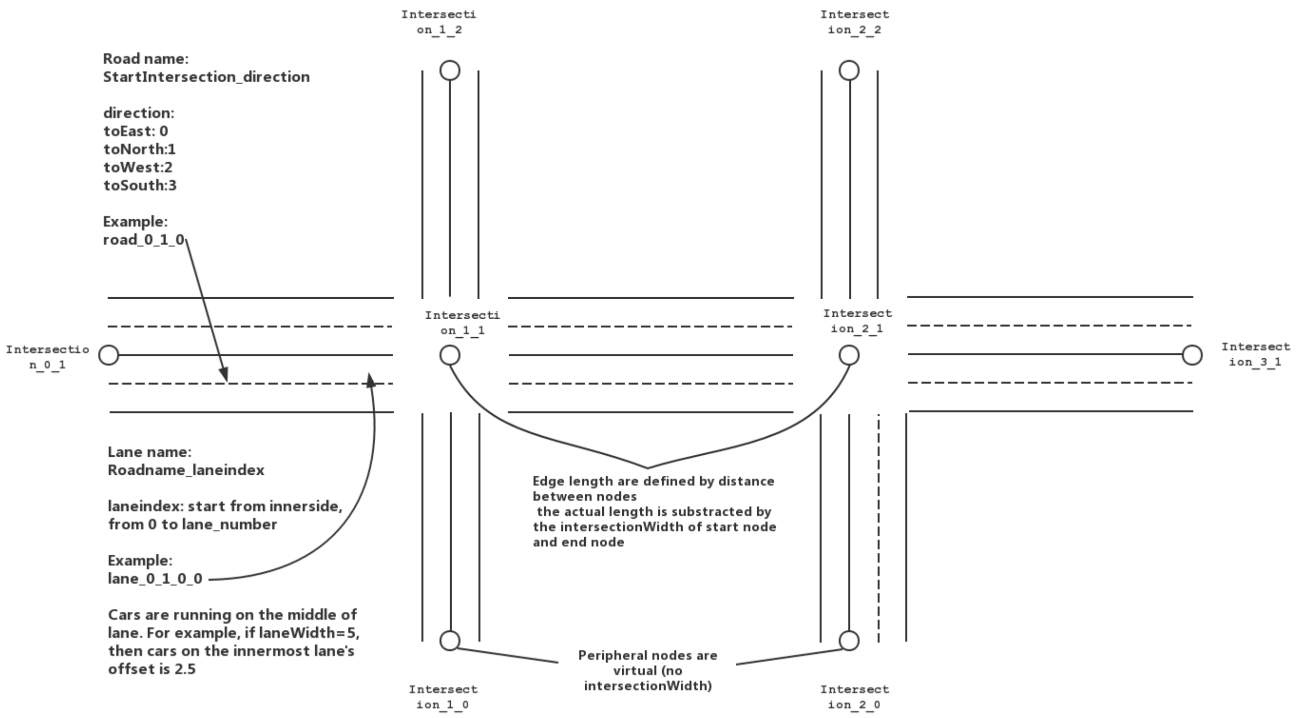

Intersection_1_2
Intersection_2_2
Road name:
StartIntersection_direction
direction:
toEast: 0
toNorth:1
toWest:2
toSouth:3
Example:
road_0_1_0
Intersection_1_1
Intersection_2_1
Intersection_0_1
Intersection_3_1
Lane name:
Roadname_laneindex
laneindex: start from innerside, from 0 to lane_number
Example:
lane_0_1_0_0
Cars are running on the middle of lane. For example, if laneWidth=5, then cars on the innermost lane's offset is 2.5
Edge length are defined by distance between nodes
the actual length is substracted by the intersectionWidth of start node and end node
Peripheral nodes are virtual (no intersectionWidth)
Intersection_1_0
Intersection_2_0

Figure 4: graph of mean reward over training iterations for our MAPPO implementation

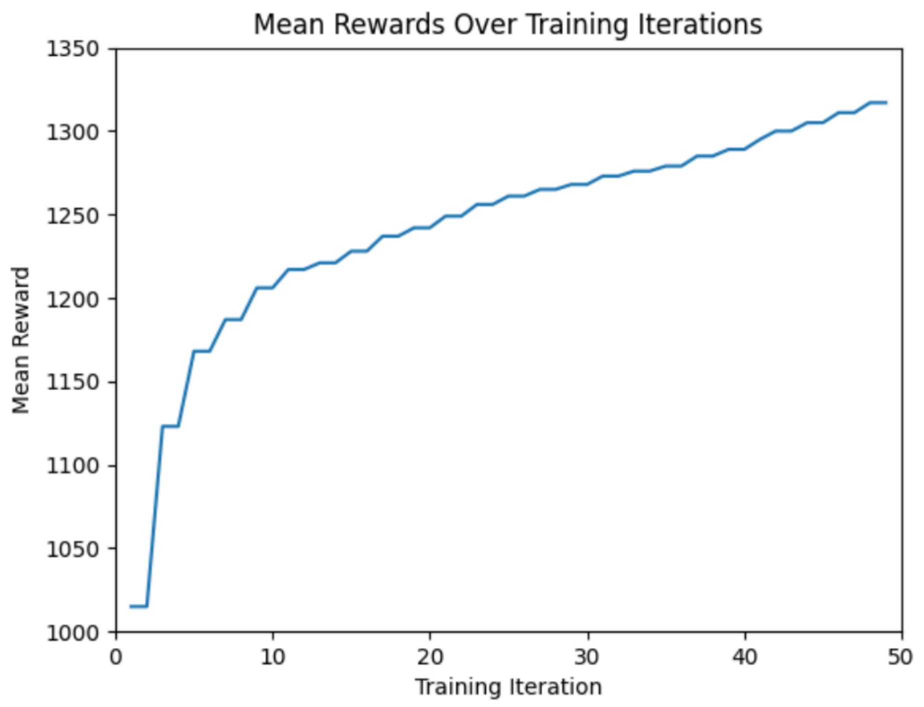


Figure 5: diagram of components of our technical approach

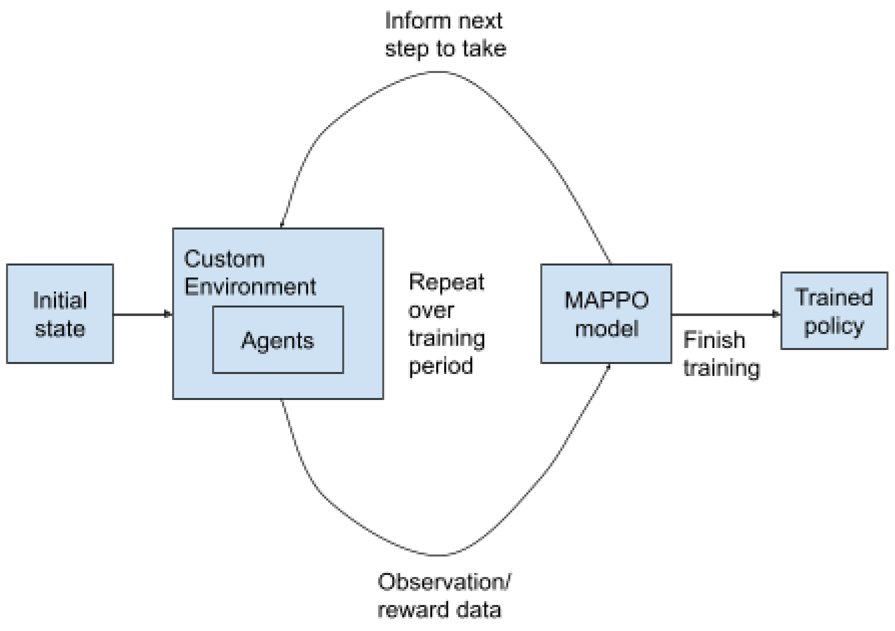
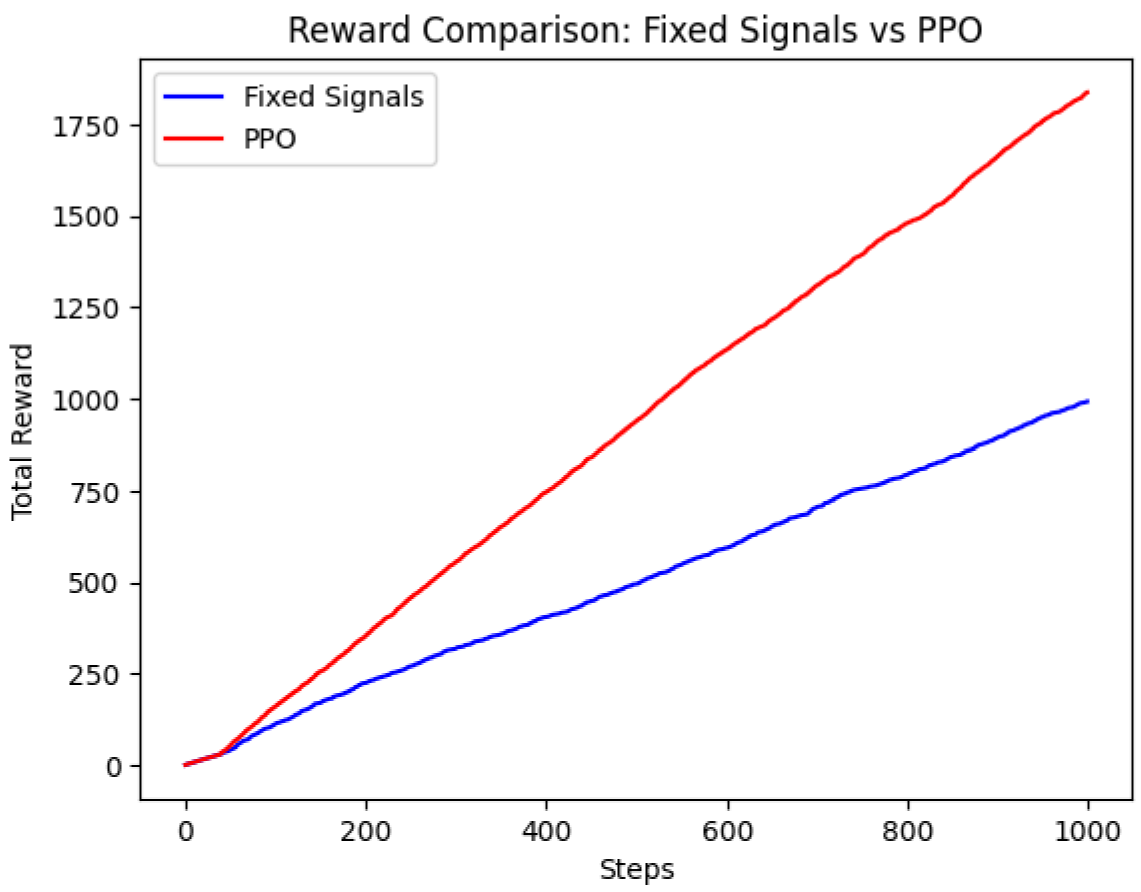


Figure 6:



9. Individual Contributions

a. Derek Xu (31628459)

For the project proposal, I mainly reviewed the final proposal that other team members wrote and added some content I felt were missing, particularly the description of the dataset and a diagram of our proposed approach. When implementing the code for the PPO model, I mainly focused on experimenting with publicly available PPO code from Hugging Face and the PPO implementation from stable_baselines3 to train our model. Specifically, I played a key role in refactoring the Hugging Face PPO code and tailoring it to our project. However, I eventually had trouble tailoring the code as it was too complex, so I tried using stable_baselines3's PPO implementation to perform the training. I wrote code for training our model with stable_baselines3's PPO implementation, and tried running it with our environment.

In addition, I have also contributed significantly to the final presentation and report as well. For instance, I created the vast majority of the final presentation, except for the description of the RLlib approach we ended up using since my other group members focused more on it. For the final presentation and report, I also created a visualization of the mean reward of our model as training progressed. In the report, I contributed significantly to writing the introduction, Related Work, Datasets, and Technical Approach sections, which I worked on with another group member. I also made some changes to the diagram I included on the proposal and included it in the final report to help describe our technical approach.

b. Rohan Mistry (23895536)
For the project proposal, I researched various ideas our group could potentially work on and decided on the traffic control systems subject. I did significant research for the project proposal, gathering over ten scholarly articles from various sources and reading through previous approaches to understand how others have approached this problem and how we could do so a bit differently. I wrote the Project Summary, Problem Definition, Proposed Technical Approach, Datasets, Experiments and Evaluation, and Software sections of the initial project proposal. I also created the proposal presentation that we used for our project proposal presentation, inserting information from the written proposal and finding visualizations online to help describe the problem. I researched existing solutions (both RL-related and without RL) and the limitations of each before settling on MAPPO as the ideal solution for the problem given the parameters and constraints of the project. I also looked into two different traffic simulators (CityFlow and SUMO-RL) and decided on CityFlow as the optimal tool for us.

To get started on the actual project, I created our group GitHub repo and installed the CityFlow API so that we could configure our environment to their formatting. I worked on setting up our custom environment and finished our initial implementation of the observation space and reward function with dynamic normalization to account for scaling grid sizes. We ran into a lot of errors with regards to debugging the environment and ensuring it would be compatible both with CityFlow's API and with our eventual PPO implementation. I spent a couple days debugging and making sure the environment was in a good state before moving on. Our first PPO implementation used the stable_baselines3 library but that did not allow for multi-agent setups so I implemented a training function using the Ray RLlib library which was designed specifically for such scenarios. After Harry fixed some issues with the reward and step functions of our environment, I began training our model on several iterations and printing results to log files to track our progress. I collaborated with Harry on the hyperparameter tuning process which was very difficult because it required significant compute power and many hours. Once we finished training the model, we saved our trained model and I implemented an evaluation function for a single episode over thousands of steps and tracked the results using CityFlow's online simulation replay tool.

For the final presentation, I worked on the Research and Development, Results and Analysis, and Reflection and Learning slides because I contributed significantly to the code implementation and training of the model. For the project submission, I created the sample Jupyter notebook to allow for simple usage of our pre-trained model. I also saved our basic data results to .csv files for easy access. For this final report, I contributed to the Data Sets section by describing the minutiae of CityFlow's API and the simulated data we used to train our model, as well as the Technical Approach and Software sections with most of the code details and the Conclusion section with the research lab prompt.

c. Harry Leung (29279838)
For the project proposal, I contributed by reviewing and providing feedback on the technical approach outlined by Rohan and conducted preliminary research on the planned implementation, including investigating potential challenges we could face in integrating reinforcement learning into traffic

simulation environments. This research allowed me to contribute meaningfully to our group discussions and ensured we were prepared to address questions during the proposal presentation.

During the implementation phase, I played a key role in improving and debugging the custom environment setup. After Rohan implemented the initial environment, I refined the step function to better align the reward structure with our project goals and ensured accurate state transitions. I resolved several compatibility issues between the environment and the stable_baselines3 PPO implementation, particularly related to state and action space normalization. When we transitioned to Ray RLlib for multi-agent support, I reviewed our environment's observation and action spaces to ensure compatibility with RLlib's requirements.

I played a significant role in designing, implementing, testing, and iteratively refining the reward function. By experimenting with and modifying the coefficients for waiting time, queue length, and throughput, I optimized the function to generate meaningful rewards that effectively guided the agents during training. Additionally, I developed scripts to visualize traffic flow and analyze key metrics from CityFlow simulations. These visualizations provided valuable insights into our model's performance and helped us identify areas for improvement. When the initial training results did not meet our expectations, I collaborated with Rohan to debug the setup, analyze training logs, and explore hyperparameter tuning strategies.

For the final presentation, I presented the Research and Development and Results and Analysis sections. Since I was significantly involved in refining the environment and reward function, I was the most viable person to explain the adaptations we made during implementation and the process for obtaining and visualizing the results. I also highlighted key insights from our iterative testing process.

In the final report, I contributed significantly to the Experiments and Evaluation section, due to my involvement in the testing and debugging phases of our process. I documented the various iterations and versions we tested, compared them to baseline values, and described the insights gained from these comparisons. Beyond my section-specific contributions, I proofread and provided feedback on other sections written by Derek and Rohan to ensure clarity and cohesion throughout the report.