ELEN4002A/ELEN4012: EIE INVESTIGATION PROJECT. MULTI-AGENT REINFORCEMENT LEARNING FOR TRAFFIC LIGHT CONTROL (23P82)

Group 23G31: Bryce Grahn (2138347)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: This investigation project aims to address traffic congestion in urban areas by leveraging reinforced learning techniques to optimize traffic light timings. Traditional methods are either inefficient, static, expensive, unreliable or a combination of the aforementioned points. This paper formulates an optimization problem with the objective of minimizing the average vehicle wait time under a cost constraint, and proposes using a traffic simulator to train a model that acquires real-time data from GPS-based applications like Waze or Google Maps, enabling it to learn and optimize traffic light timings. The project was implemented in six stages: research, design of single and multi-agent system models, data collection and map integration, creation of a camera, GPS, and advanced observation environment, optimization and hyper-parameter tuning, and concluding with result generation and documentation. The optimal solution was solved using a PPO algorithm where the environment was simulated using SUMO and the agent trained in a PettingZoo gym with Sumo-rl. Training was both improved and made faster using Optuna and SuperSuit in tandem with a custom observation and reward class. The simulation results demonstrate that the proposed GPS model surpasses existing implementations in 72% of cases, with a potential for a 25.7% performance boost to 97.6% in an idealistic environment. In addition the proposed engineering solution is a fraction of the cost and exhibits great flexibility and scalability in both the type and number of intersections.

Key words: Multi-agent reinforced learning (MARL), Proximal Policy Optimization (PPO), Fixed-time traffic signal controllers (TSC), Markov decision process (MDP), Simulation of Urban Mobility (SUMO), Global positioning system (GPS). Adaptive traffic signal control (ATSC).

1. INTRODUCTION

In recent years, traffic congestion has become a significant issue in urban areas and can be attributed to factors such as population growth, greater reliance on personal vehicles, urbanization, and insufficient infrastructure development. Cities like London and Dublin have seen travel times increase by 5.3% to 5.9%, and an average travel speed of 17 km per hour well below the 48-kilometer speed limit [1].

The increase in transportation demand can be met by providing additional capacity to the transportation infrastructure. However, expanding infrastructure to meet growing transportation demands isn't always economically or socially viable. Thus, optimizing the use of the existing infrastructure is essential. One effective solution is to employ adaptive control of traffic signal controllers (ATSC).

Motivated by the above observations, this paper formulates an optimization problem with the objective of minimizing the average vehicle wait time under a cost constraint, and proposes using a traffic simulator to train a model that acquires real-time data from GPS-based applications like Waze or Google Maps, enabling it to learn and optimize traffic light timings.

The paper is structured as follows: Section 2 presents a literature review. Section 3 introduces the project solution. Section 4 outlines the project's design, and section 5 focuses on project management. Section 6 details the results, followed by a critical analysis in section 7. Recommendations are made in section 8, and section 9 concludes the paper.

2. LITERATURE REVIEW

2.1 Traditional implementations

Traditional implementations involve fixed-rule algorithms. These Fixed-time traffic signal controllers (TSC's) utilize predetermined fixed-time intervals for each phase, following a consistent cycle [2]. However, they are unable to meet current traffic-stochastic needs or deal with unanticipated traffic circumstances. Adaptive traffic control or ATSC is a traffic management strategy in which traffic signal timing changes, or adapts, based on actual traffic demand [3]. ATSC is often used in tandem with camera systems where max pressure or greedy algorithms are implemented [4]. However, the effectiveness is highly dependent on the position and reliability of the sensors.

2.2 Existing solutions

The ATSC problem can be solved using vision based algorithms [3]. However, these models demand a complete traffic understanding and require more improvements to provide optimal timings. Hence, multi-agent reinforcement techniques are the current leading approach in managing vehicles within road networks. Q-learning is an off-policy algorithm which does not require a pre-defined environment model. Abdulhai 2003 et al displayed promising results with their Q-learning green light district simulator [5]. However, the results showed a performance degradation for over-saturated conditions. In the work of Wunderlich et 2008, a "longest-queue-first-maximal-weight-matching (LQF-MWM) method" outperformed existing solutions at the time [6]. The

work was mostly limited to two isolated intersections and hence, lacked robustness. Arel et al 2010 proved the advantages of MARL-based control over LQF-based methods [7]. Aziz et al investigated information sharing with their proposed R-Markov Average-Reward-Technique-based RL (RMART) algorithm that performed well in over-saturated conditions but was limited to small scale networks [6]. The KS-DDPG deterministic algorithm showed promising performance increases over RMART in large scale networks [8]. Antes et al 2022 improved agent performance with a hierarchical information sharing scheme [9]. The aforementioned methods require improvements in terms of accuracy, observations, rewards, costs, results, scalability, and performance.

2.3 Centralized vs Decentralized vs Curriculum

In centralized learning, a central policy is formulated from individual agents' experiences and shared. In decentralized learning, individual agents independently acquire knowledge without engaging in information sharing [10]. Centralized learning reduces learning effort whereas decentralized offers both simplicity and straightforward scalability. Centralized learning is often avoided because it lacks robustness (central point of failure) [10]. Curriculum refers to a structured approach where a sequence of tasks is ordered by complexity, and knowledge is transferred between tasks [10]. The Curriculum approach did not demonstrate a noticeable improvement as both non and curriculum based approaches converged to a similar solution.

2.4 PPO vs DQN

PPO defines a term called ‘advantage’ as A(s,a)=Q(s,a)-V(s) where V(s) is approximately the same value you would get by averaging all the Q-values of every action in a given state. Policy based algorithms can perform better compared to value-based algorithms in stochastic environments and have shown to be more stable during training [11].

Table 1: PPO vs DQN comparison

Feature	PPO	DQN
Learning Approach	Learns a policy	Learns a Q-function
On/Off Policy	On-policy	Off-policy
Key Techniques	Clipped surrogate objective and generalized advantage estimation	Experience replay and target network
Action Space Handling	Discrete and continuous action spaces	Can only handle discrete action spaces
State Space Suitability	Suitable for problems with large or infinite action spaces	More suitable for problems with large or high-dimensional state space

3. PROJECT DESCRIPTION

3.1 Problem statement

A significant challenge in the optimization of traffic light controllers lies in the highly restrictive nature of the problem, since minimum and maximum green times need to be observed, driver reaction times must be considered, and the control policy needs to be fair for all traffic directions. Additionally, deploying monitoring cameras at every intersection for real-time data collection is expensive and impractical in certain regions. The proposed solution must demonstrate exceptional adaptability, capable of effectively managing multiple intersections, each with an unknown structure. To promote scalability and compatibility with popular GPS applications like Waze, Google maps, or TomTom, it is imperative that the observation and reward space remain simple.

3.2. Proposed solution

The proposal is as follows: Simplify complex traffic signal control through the use of multi-agent reinforcement learning (MARL). Ensure scalability and robustness with a decentralized implementation. Minimize vehicle wait times, maximize travel speed and throughput, and reduce traffic stops by promoting traffic signal coordination. Optimize traffic light timings using a PPO algorithm, simulating the environment with SUMO and agent training in PettingZoo. Enhance training efficiency with Optuna and SuperSuit, aided by a customized observation and reward systems. Reduce costs by acquiring observation data from GPS-based applications like Waze, Google Maps or TomTom.

3.3 Success criteria

Successful execution of the project is separated into several sections. Firstly the successful implementation of a PPO based MARL model that is both scalable and independent from intersection types. Secondly, introducing a solution that offers an effective cost benefit over camera based observations. Success is confirmed when the proposed solution outperforms existing fixed control, and algorithmic camera based implementations making use of max pressure and greedy algorithms. Finally, all system parameters, objectives, and system models should be carefully justified, and results should be critically analyzed.

Section 4 details the detailed design procedure of the aforementioned solution.

4. DETAILED DESIGN

SUMO-RL provides a simple interface to instantiate reinforcement learning environments with SUMO for traffic signal control. It supports multi-agent RL and offers easy customization of state and reward definitions. Additionally, it integrates with popular RL libraries such as SB3 and RLlib. The system employs the TraCI API to retrieve information and control traffic lights within SUMO. SuperSuit is used to pad the action and observation space, and vectorize the PettingZoo environment for each agent in preparation for Multi-agent networks [12]. To replicate real-world scenarios, actual map data is imported using the OpenStreetMap tool provided by SUMO. To simulate real-world traffic data within the simulation, an approximation of traffic density is generated using the randomTrips.py script, which allows for the definition of parameters like insertion rates per hour. This approach is consistently applied across all seven selected maps present in Figure 1 of Appendix D.

4.1 Parameter justification

Table 2: Parameter justification

Parameter	Value	Justification
Camera range	35m	Smart traffic cameras in existing literature have an operation distance of 35m.
Cars visible using GPS	77%	According to, 77% of smartphone owners regularly use navigational apps.
Time between actions	8 s	Three seconds of yellow time and a minimum of five seconds of green time yields a total of 8s before the agent chooses another action. This is a consequence of the driver reaction time and vehicle deceleration rate
Seconds of simulation	3600s	This one-hour traffic flow duration effectively represents all traffic conditions, from lighter traffic to peak rush hour scenarios.
Min yellow time	3 s	The Federal Highway Administration's Manual recommends the yellow interval be between three and six seconds. A vehicle decelerating at 4.5m/s^2 from 45 km|h will take 2.77 s < 3 s
Driver reaction time	0.75 s	How long it takes for a driver to move from the accelerator pedal to the brake pedal in response to a situation
Vehicle deceleration rate	4.5 ms^{-2}	A driver can slow down at this rate without loss of control of the vehicle in the process

4.2 Markov decision process

A signalized intersection comprises incoming and outgoing roads, each consisting of one or more lanes. The intersection is assigned a set of phases, Φ, where each phase, φ ∈ Φ, corresponds to a specific traffic movement through the intersection. These phases can be in conflict if they cannot be enabled simultaneously as seen in figure 1. The signal controller's task is to optimize a long-term objective by enabling non-conflicting phases at each time step. The MARL based controllers are modeled as a Markov Decision Process (MDP) with a state and action space defined by incoming traffic conditions and enabled phases respectively. The state or observation depends on the assumed sensing capabilities mentioned in section 4.5.

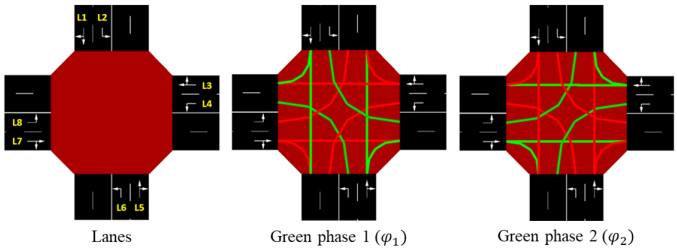


Figure 1: Two examples of different intersection phases

4.3 Design specifications and architecture

To review the selected design specifications, please consult Table 1 in Appendix D, and for an overview of the architecture, refer to Figure 5 in Appendix D. Detailed information about the environment and traffic-signal code is provided in Sections 1 and 2 of Appendix H.

4.4 Training architecture

Reinforcement Learning consists of an agent and an environment. At time step t, the agent is in state $s_t \in S$, performs action $a_t \in A$, transitions to state s_{t+1} and receives a reward $r_t = R(s_t, a_t, s_{t+1})$. The agent's actions are governed by its policy $\pi(a|s)$, which is the probability of performing action 'a' in state 's'. PPO defines a term called 'advantage' as $A(s,a) = Q(s,a) - V(s)$ where V(s) is approximately the same value as averaging all the Q-values of every action in a given state. Therefore, the advantage can be described as an estimate for how good an action is compared to the average action for a given state. PPO maintains two separate policy networks. the current policy to be refined, denoted $\pi_\theta(s|a)$, and and a previous policy established after collecting certain experience samples, represented as $\pi_{\theta k}(a|s)$. The algorithm introduces incremental adjustments to the prior policy to avoid making drastic changes throughout the learning process. This is achieved using a clipped objective function that discourages large policy changes when the ratio of the current policy and old policy ($r_t(\theta)$) falls outside a defined "comfort zone" [3]. For a visual representation of the training loop, refer to Figure 4 in Appendix D..

4.5 Observation space, action space, and reward function

Observation space: [current phase ID of the intersection (one hot encoded), the normalized queue lengths of each incoming lane (waiting vehicles), the normalized occupancy of each incoming lane, and the normalized average speed of each incoming lane]. See figure 11 of section 6 as validation of this observation's performance.

Action space: [$\{\varphi_1, \varphi_2, \varphi_3, ..., \varphi_i\}$, α]. Where φ represents an allowable phase (yellow, red, or green) for road i and α represents the duration in seconds.

Reward function: [Difference in wait time + Difference in speed]. See figure 2 of section 6 as validation of this reward's performance.

Please refer to Appendix I and J, and code in sections 9 and 10 of Appendix H for the full observation and reward spaces respectively.

4.6 Benchmarks

1. Fixed Timings: Traffic lights adhere to predetermined timing sequences for each phase.
2. Greedy Control: Initiates the green phase for incoming lanes with the longest queue length, aiming to optimize local throughput using camera observations.
3. Max Pressure Control: Initiates the green phase for incoming lanes with the highest joint pressure, striving to enhance global throughput using camera observations.

Please refer to Figure 2 in Appendix E for a visual representation.

5. PROJECT MANAGEMENT

The tasks that were carried out during the project were allocated amongst both group members as depicted in Table 1 of Appendix A. The experiences that were encountered during the project are also documented in Appendix A. Table 3 shows the timely completion of each project milestone. For further details see the work breakdown structure and Gantt chart in appendix B. See Appendix E and F for cohort and individual meetings respectively.

Table 3: Project milestones

Milestone	Expected date	Executed on time
Research and initial setup	10/09/2023	☑
Single and multi-agent training	17/09/2023	☑
Metrics and benchmarks	24/09/2023	☑
Camera, GPS, and Ideal environments	01/10/2023	☑
Resolve issues and tune hyper-parameters	08/10/2023	☑
Generate results	15/10/2023	☑
Report writing	27/10/2023	☑

6. RESULTS

Figure 3: Map of Cologne 8

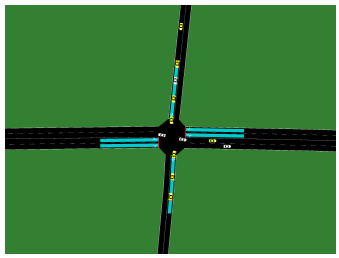


Figure 4: Map of a single intersection

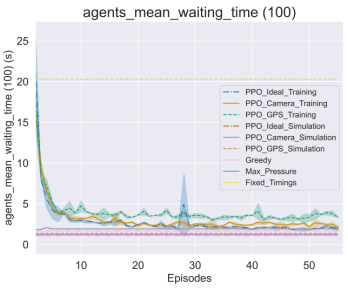


Figure 5: Agent mean waiting time over 100 seconds of travel

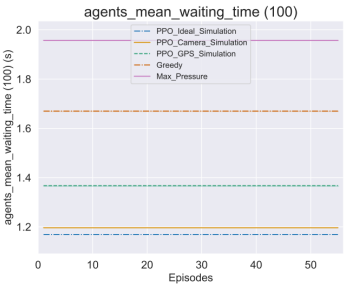


Figure 6: Magnified section of figure 6

Figure 7: Average vehicle speed

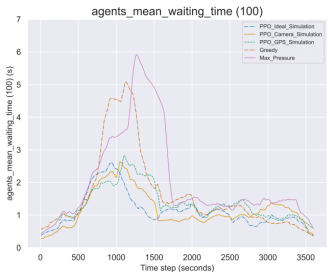
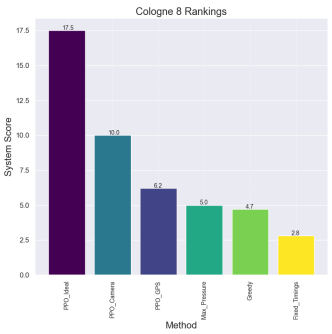


Figure 8: The average vehicle waiting time over 1 hour



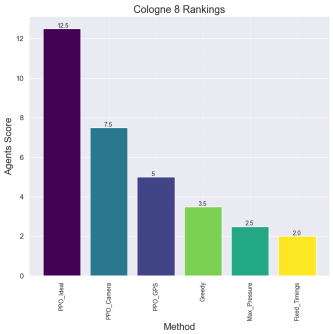


Figure 9: System rank Figure 10: Agent rank

Table 3: Rankings across the 7 chosen maps

Map	1st	2nd	3rd	4th	5th	6th
Beyers	Ideal	Cam	Max P.	GPS	Greedy	Fixed
Ing1	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Ing7	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Ing21	Ideal	Cam	Max P.	Greedy	GPS	Fixed
Col1	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Col3	Cam	Ideal	GPS	Greedy	Max P.	Fixed
Col8	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Final	**Ideal**	**Cam**	**GPS**	**Greedy**	**Max P.**	**Fixed**

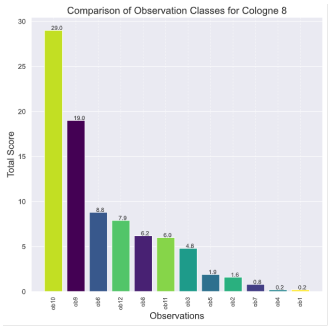


Figure 11: Comparing observations

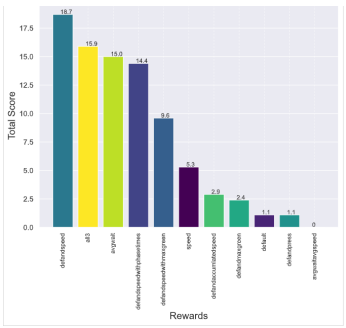


Figure 12: Comparing rewards

6.1 Brief overview

This section conducts a performance analysis of the proposed system, emphasizing mean waiting time and speed. Figure 3 depicts a section of Cologne 8, while Figure 4 zooms in on a single intersection within Cologne 8. Yellow cars denote observables, white cars are not. Blue lanes signify camera-monitored areas around the intersection, with the ideal scenario covering the entire map. To ensure data integrity, results from agent-monitored intersections are isolated in Figures 3 to 6, avoiding distortion from stop streets and four-way stops. Figure 4 offers a closer look at Figure 3 for model differentiation, and Figure 6 plots the average vehicle waiting times over one hour. A total of 12 metrics rank each model or benchmark for Cologne 8, as seen in Figures 10 and 11. Table 3 compares the seven chosen maps and their ranks. Figures 11 and 12 compare 12 observation classes and 11 reward functions for Cologne 8. Please refer to Appendix C to view additional results.

7. CRITICAL ANALYSIS

7.1 Overall functionality

Introducing a reward function for both wait time and speed difference enables the model to reduce average waiting times for each car (Figure 5 and 7), leading to fewer stops and improved travel throughput across lanes. The model achieved a promising 1.4-second wait time compared to 20 seconds in the fixed-time scenario over 100 seconds of travel. Notably, it demonstrated quicker recovery with higher car volumes compared to fixed-time or camera-monitored systems between 700 and 1800 seconds (Figure 9). In addition, Gps outperforms the fixed average speed of 17 km with a speed of 30 km in figure 7.

Mean waiting time and average speed are among the 12 metrics used to assess each model. These metrics cover a range of factors, including the number of cars, stopped cars, and metrics specific to agent-controlled intersections and the entire system. Figure 9 and 10 provide a visual representation of these rankings. Notably, the top three trained models outperformed all three benchmarks across all 12 metrics.

Observation 10, comprising 4 observations, achieved the best results. This is advantageous as it prioritizes simplicity and scalability, which a smaller observation space supports. Interestingly, the next best observation encompasses every possible scenario, which would enhance model robustness but increases complexity and costs in managing the observation space. Figure 15 illustrates the different rewards, with the one considering speed and wait time differences being identified as the most effective.

The ideal RL model placed first on 6 out of 7 maps and the GPS solution outperformed each benchmark in 18 out of 21 comparisons. For performance margins in comparison to fixed time control. Ideal reduced the waiting time by 91.05%, camera reduced the waiting time by 90.91%, and GPS reduced the waiting time by 86.08%.

7.2 Robustness and reliability

The proposed engineering solution consistently outperforms Fixed timings across seven different maps with varying intersections and traffic conditions, achieving a 100% success rate. It also surpasses max pressure and greedy methods 72% of the time. In an ideal scenario, the model has the future potential to outperform existing benchmarks 97.6% of the time. The 72% lower performance rate can be attributed to limited training time for larger maps like Ingolstadt21 and Beyers. Unlike a single user's cellphone, which only provides data for one car while at an intersection, a failed camera can leave an entire road unmonitored for several days. Hence, GPS is more reliable.

7.3 Assumptions and Limitations

1. Swift driver reaction time, 2. Minimum 5-second green phase, 3. Minimum 3-second yellow phase, 4. Expensive initial training costs, 5. 1-second SUMO update resolution, 6.

77% driver usage of Waze or Google Maps, 6. Full adherence to traffic laws (no speeding, collisions, or interactions with pedestrians).

7.4 Cost analysis

For 11 GPS-monitored intersections, the total cost comprises R13,500 for training and R88,000 for installing encrypted microcontrollers with antennas on each robot. Each intersection communicates with a dedicated server capable of online learning at a monthly subscription cost of R1,900, resulting in a total cost of R101,500. In contrast, the total cost for 11 camera-monitored intersections is R14.3 million. In summary, it would take 56 years for the proposed solution to surpass the upfront installation costs of a camera-based system.

8. RECOMMENDATIONS AND FUTURE IMPROVEMENTS

The engineered solution outperforms existing implementations across twelve metrics and eight maps and is especially true for a perfectly observable environment. However, it is limited in its resolution and driver reaction time. As such future improvements are motivated to include online learning for greater flexibility, a curriculum approach for reducing training costs, a fall-back/fail-safe system, an increase in training resolution, self-driving cars, and a centralized approach for reward sharing and cross agent communication. The model also relies heavily on the simplifications made and is advisable to reduce the number of assumptions and have a more complex observation and action space should the technologies allow it.

9. CONCLUSION

This paper investigated a number of strategies for optimizing traffic light timings, namely TSC's and ATSC's with the purpose of reducing traffic congestion in urban areas. The proposed engineering solution successfully met all design, objective, and success criteria beyond expectations. In summary, the model demonstrates outstanding performance, adaptability, and robustness across different regions, intersection types, and volumes of traffic data, at a reduced cost.

ACKNOWLEDGEMENTS

The author would like to thank the project supervisor, Prof Vered Aharonson, and team member Michael Rolle for their help throughout the length of the project.

REFERENCES

[1] T. Tom. (2023) "Traffic-index-rankings", *TomTom*, vol. 1, no. 1, pp 1

[2] X. Hou, L. Chen, J. Tang and J. Li. (2022) "Multi-Agent Learning Automata for Online Adaptive Control of Large-Scale Traffic Signal Systems," *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, Rio de Janeiro, Brazil, pp. 1497-1502

[3] Reza, Selim, et al. (2023) "A Citywide TD‑ Learning Based Intelligent Traffic Signal Control for Autonomous Vehicles: Performance Evaluation Using SUMO." *Expert Systems*, vol. 1, 5, pp. 2–18

[4] M. Rolle. (2023). "A decentralized multi-agent reinforcement learning approach", *Investigation Project*, University of the Witwatersrand

[5] Abdulhai, B., Pringle, R., & Karakoulas, G. J. (2003). "Reinforcement learning for true adaptive traffic signal control". *Journal of Transportation Engineering*, 129(3), pp. 278-285

[6] Wunderlich, R., Liu, C., Elhanany, I., & UrbaniklI, T. (2008). "A novel signal-scheduling algorithm with quality-of-service provisioning for an isolated intersec-tion". *Intelligent Transportation Systems*, 9(3), pp. 536-547.

[7] Arel, I., Liu, C., Urbanik, T., & Kohls, A. G. (2020). "Reinforcement learning-based multi-agent system for network traffic signal control". *IET Intelligent Trans-port Systems*, 4(2), pp. 128-135.

[8] Li, C., & Xu, P. (2021). "Application on traffic flow prediction of machine learning in intelligent transportation". *Neural Computing and Applications*, 33(2), pp. 613-624

[9] Antes, T. O., Bazzan, A. L., & Tavares, A. R. (2022). "Information upwards, recommendation downwards: Reinforcement learning with hierarchy for traffic signal control". *Procedia Computer Science*, 201, pp. 24-31.

[10] T. Kaur and S. Agrawal. (2014). "Adaptive traffic lights based on hybrid of neural network and genetic algorithm for reduced traffic congestion," Chandigarh, India, pp. 1-5

[11] Brian. (2022). "Introduction to Multi-Agent Reinforcement Learning." *www.youtube.com*, The MathWorks, Inc.,

[12] L. Alegre. (2019) "Sumo-rl", *Sumo-rl 1.43 documentation*, vol. 1, no. 1, pp 1

[13] d. gov. (2020). "Implementation costs for automated camera systems" *U.S. Department of Transportation.*

APPENDIX A: GROUP WORK REFLECTION

1. Introduction

In any engineering project where more than one person is involved, full participation of all the parties can often mean the difference between success or failure. Generally, mis-understandings between group members can prevent the team from delivering an outcome that truly mirrors their collective capabilities. In this project, both partners acquired essential skills in project management and how to work effectively as a team.

2. Team work

In any project, it is important to understand both the strengths and weaknesses of each group member. It makes it that much easier to split up the work amongst team members that are more suited and inclined to deliver a higher standard of work. Because each member knew of the other it was much easier to accelerate the initial phases of the project to a point where results were obtained by the end of week 2. From the onset, expectations and working hours were discussed to prevent future misunderstandings. Often it is essential to assign roles, and have a dedicated leader. However, both members have equal experience and to avoid micromanaging, ensure shared accountability, and faster decision making, no team leader was allocated. In addition, understanding each other on a personal level and why certain behaviors were demonstrated was a key aspect in maintaining a positive and productive working environment.

3. Communication

Effective communication amongst team members ensures seamless collaboration and the efficient pursuit of the project's common goals. Various communication platforms such as Whatsapp, Discord, Gmail, and MS Teams were utilized. Communication was so effective that it was decided that both members would work separately from home. Google docs was used for project documentation whilst VScodes screen share feature was used to quickly and effectively resolve issues present in each other's code. Git in combination with Github was utilized as a version control system to consistently track and share changes in an agile approach.

4. Splitting up the work

From the beginning, it was discussed who would be more interested in a section of the project and is more suited to the task with their particular skill set and the workload was usually divided as such. Table 1 below shows the work division and expected working hours. The split would ensure both members were assigned equal volumes of work. In addition, group members were expected to take on different parts of the project to improve their knowledge in areas that the other group members had focused on. This proved to be a very successful strategy.

5. Challenges and how they were addressed

In a group, challenges and conflicts are inevitable. However it is important to identify these issues and address them promptly. One prominent challenge encountered was the need to vectorize the environment, enabling multiple agents or traffic light intersections to interface with conventional single-agent Reinforcement Learning (RL) methods. An issue that required resolution before commencing with the remainder of the project. Both team members demonstrated a willingness to invest extra time to find a solution. Consequently, within a matter of days, a successful resolution was brought about. Future problems were resolved in a similar manner.

6. Overall reflection and conclusion

Reflecting on group work is also an opportunity for personal growth. It fosters a healthy, productive, and innovative working environment that benefits both the parties involved and the group as a whole. Each group member displayed a superior work ethic with a plethora of skills vital to the project's success. The ease of communication facilitated a seamless and productive

environment that benefited everyone. Sharing ideas and solutions was always open, supported and encouraged. Apart from teamwork development, the project contributed tremendously to the personal growth of the author as an individual in terms of time management, communication, problem solving skills, critical thinking, the importance of reliability, communication and dedication in the project. In summary, team work between the group members resulted in the project being successfully executed.

Table 1: Workload allocation

Task	Duration (hrs)	Group member	
		Bryce	Michael
Week 1: Research and initial setup			
1. Background research	5	☑	☑
2. SUMO installation and setup	4	☑	
3. Import real-world maps	3	☑	
4. Research centralized, decentralized, and curriculum approaches to reinforcement learning	2		☑
5. Evaluate POSG and AEC model environments	2		☑
6. Research traditional Fixed-time traffic signal controllers and existing implementations	1		☑
7. Install and setup the sumo-rl interface with SUMO and PettingZoo	8	☑	☑
8. Research MARL algorithms	2	☑	
9. Evaluate MARL toolkits	2		☑
10. Research optimization techniques	5	☑	
Week 2: Train a single and multi-agent combination of intersections			
1. Define the Markov decision process	8	☑	☑
2. Create a script for single agent training	6	☑	
3. Pad the action and observation space, and vectorize the PettingZoo environment for each agent in preparation for Multi-agent networks	3		☑
4. Create a training script for a simple 2x2 grid network of intersections	7		☑
5. Implement parallel environment learning and distributed computing across multiple CPU's	1	☑	
6. Export, import, and simulate the learnt model	6	☑	
7. Setup Azure virtual machines for training	3		☑
8. Create car route and network files	2	☑	
9. Implement DQN and PPO algorithms	3		☑
Week 3: Metrics, rewards, observation spaces, and benchmarks			
1. Calculate and log training and simulation metrics such as waiting time, speed, etc	10		☑
2. Implement random and fixed control traffic light timings	16	☑	
3. Implement a custom observation and reward class	13		☑
4. Import and setup benchmark maps that cover single, corridor and region type collections of intersections	5		☑
5. Implement greedy and max-pressure controlled intersections	10	☑	
Week 4: Camera, GPS, and Ideal observation environments			
1. Develop a script to rank models, rewards, and observations based on the logged metrics.	5		☑
2. Develop a script for logging metrics related to agent-controlled intersections only	4		☑
3. Implement a camera or GPS monitored network of intersections	12	☑	
4. Implement a fully observable network of intersections (Every car and its metrics are known)	9		☑
5. Visualize camera-monitored sections and differentiate both tracked and untracked cars.	2	☑	
Week 5: Resolve issues, optimize code, and improve training			
1. Remove cars exceeding a waiting time threshold	1		☑
2. Remove cars involved in collisions	4	☑	
3. Code a script that repairs corrupted result files	2		☑
4. Tune hyper-parameters using Optuna	3		☑
5. Documentation	12	☑	☑
Week 6: Generate results			
1. Train models for each Cologne map using the various approaches (Greedy, max-pressure, random, fixed) and observation techniques (camera, GPS, ideal full observable environment)	32		☑

2. Train models for each Ingolstadt map using the various approaches (Greedy, max-pressure, random, fixed) and observation techniques (camera, GPS, ideal full observable environment)	32	☑	
3. Plot and rank results	5	☑	☑
4. Discuss the feasibility of integrating the system with Waze or Google maps	4		☑
Week 7: Report writing	30	☑	☑

APPENDIX B: PROJECT PLAN

ELEN4002A/ELEN4012: EIE INVESTIGATION PROJECT. INVESTIGATION INTO REDUCING CONGESTION WITH TRAFFIC LIGHT TIMINGS USING MACHINE LEARNING (22P82) PROJECT PLAN.

Group 23G31: Bryce Grahn (2138347) and Michael Rolle (2304928)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: This investigation project aims to address traffic congestion in urban areas by leveraging machine learning and reinforced learning techniques to optimize traffic light timings. Traditional methods often result in inefficiencies, and the project proposes using a traffic simulator to train a model that analyzes real-time data to determine optimal traffic light timings. Publicly available driving datasets will then be used to validate the model in real world scenarios. The project will be implemented in six stages, starting with researching various methodologies to find the best software and tools to use, setting up a virtual environment for training and simulation, training a single-agent model to implement at intersections independently, training a multi-agent model to attempt to synchronize multiple intersections better, optimizing the existing models and choosing the best one, and concluding with the validation of the model with real life traffic data as well as the theoretical discussion of model migration into the real world. Python, Google Collaboratory, and project management tools will be utilized. The project's success relies on dataset availability, reinforced learning effectiveness, and timely access to technologies, while facing technical limitations and time constraints. The investigation project will be completed within the designated timeline by dividing tasks and utilizing appropriate resources.

Key words: Reinforced learning, Machine learning, Multi-agent reinforced learning (MARL), Traffic light timing

1. INTRODUCTION

The management of traffic congestion in urban areas is a significant challenge that requires innovative solutions [1]. The investigation project seeks to investigate a possible solution by utilizing machine learning techniques that optimize traffic light timings. Traditional methods, such as fixed timings for traffic lights, often result in inefficiencies and delays due to the complexity of traffic patterns. Additionally, deploying monitoring cameras at every intersection for real-time data collection is expensive and impractical in certain regions.

To overcome these limitations, the project proposes a more modern approach that leverages publicly available driving datasets, like Waze, to train a machine learning model. These datasets contain driving conditions such as congestion, traffic, accidents, with some offering continuous updates in real time. Giving the model the potential for live updates on traffic congestion in the same way that physical cameras would.

This project will be implemented in six stages. The first is research of various methodologies to find the best software and tools. The second is setting up a virtual environment for which a model can be trained and simulated using reinforced learning. The third stage involves training a single-agent model that can be applied at each intersection independently. A multi-agent model will then be trained in the fourth stage to attempt to improve synchronization and coordination between intersections. Centralized and decentralized networks will be tested for the multi-agent model. The fifth stage involves refining, optimizing and comparing the models to find the best implementation. The final stage will then be to integrate real world traffic data into the simulation to validate the chosen model against traditional implementations. A discussion will then be held to see how the model would theoretically be implemented into the real world.

1.1. Background

Researchers are continuously trying to optimize how traffic lights operate to reduce congestion as much as possible. The recent emergence of ‘smart traffic lights’ that utilize machine learning are outperforming traditional implementations [2].

Traditional implementations involve fixed-rule algorithms [3] and include:

1) Fixed-time control: This is the simplest and most common approach. It uses predetermined fixed-time

intervals for each phase and allows a fixed cycle to take place.
2) Max-pressure control: This aims to maximise the throughput capacity by prioritizing green-time for lanes that have the highest number of vehicles. This is achieved using physical sensors such as cameras or inductive loops.
3) Greedy control: This algorithm selects the phase combination that has the maximum joint queue length and highest number of approaching vehicles and activates those lanes first. This approach also requires physical sensors for traffic detection.

It has been shown that these traditional methods have been outperformed by recent works that use reinforcement learning to train machine learning models that can control traffic lights [3]. These models still require access to live traffic data to make dynamic changes. Most implementations opt to use physical sensors such as cameras at intersections. These sensors are costly and would require a lot of resources to install at every intersection [4]. The aim of this research project is to provide an affordable solution that retrieves live traffic data from an external source (such as an API from Waze or vehicles' GPS information) and is able to recognize traffic patterns in an area to reduce congestion through dynamic decision making.

1.2. Literature review

There are a variety of machine learning methods that people have attempted to implement to reduce traffic congestion.

One of these methods involves creating a traffic predictor model that uses regression techniques or artificial neural networks on large and complex datasets to identify patterns and relationships in traffic. It uses past traffic data of a particular area to predict future traffic conditions based on recognizable trends [5]. These traffic predictors eliminate the need for physical sensors at intersections and allow traffic light controllers to make decisions based on their predictions. These predictions are, however, still predictions and may not give an accurate reading of the true state of traffic at a point in time.

The preferred method discusses the application of reinforced learning (RL) for traffic light control. Realistic traffic data is used in a traffic simulator to train a machine learning model using RL techniques. SUMO (Simulation of Urban MObility) appears to be the most comprehensive and widely used software to do this. SUMO supports the import of real urban infrastructure using OpenStreetMap (OSM) to allow the training of a model for a specific real-world location [3]. RL methods work in real time through trial and error and therefore do not require a previous dataset for training.

Single agent models will have independent agents at each intersection that make decisions solely on the traffic state at that intersection. Multi-agent models allow for these agents to be interconnected and update their policies based on other agents' policies. Multi-agent models are more complex to implement but allow for better synchronized traffic flow between intersections. For multi-agent models, agents will be able to communicate with each other through either a centralized or decentralized network.

There are multiple different RL methods that have been widely applied to traffic light control optimization. Some notable approaches include:

1) Q-learning: This models interconnected intersections as a graph and uses tabular Q-learning to find optimal traffic light switching times based on vehicle waiting times.
2) Deep Q-learning: Uses deep neural networks to approximate the Q-values in the Q-learning framework. It has shown to outperform traditional Q-learning in various RL tasks.
3) Proximal Policy Optimization (PPO): A policy optimization method that iteratively updates the policy by maximizing a surrogate objective. It has demonstrated good sample efficiency and stability in RL tasks.

To integrate these RL models into the real world, the most commonly used approach is to use physical sensors placed at intersections to update the controllers on the current traffic state.

2. PROJECT DESCRIPTION

2.1. Problem statement

This research project addresses the challenge of managing traffic congestion in urban areas. Traditional methods, such as fixed timings for traffic lights, often result in inefficiencies and delays due to the complex nature of traffic patterns. Deploying monitoring cameras at every intersection for real-time data collection is expensive and impractical in certain regions [4]. To overcome these limitations, the project proposes using publicly available live driving datasets to train a machine learning model that optimizes traffic light timings. By leveraging reinforced learning, the model can analyze driving data, including congestion and accidents, to determine the most effective timing for traffic lights, thereby reducing congestion and improving traffic flow [6].

While the physical implementation involving a central server and wireless chips on traffic lights is theoretical and outside the project's scope, this research aims to lay the groundwork for practical applications. By employing machine learning and real-time driving datasets, traffic congestion can be reduced without the need for costly monitoring cameras. The project strives to improve traffic management, enhance traffic flow,

and address the limitations of traditional methods, contributing to efficient transportation systems in urban areas.

2.2 Requirements and success criteria

To accomplish this investigation, essential resources include access to public driving datasets, machine learning libraries and frameworks (such as SUMO-RL and PettingZoo), computing resources, and simulation software (SUMO). The project will be implemented in six stages: researching various methodologies to find the best tools, setting up a virtual environment for model training and simulation, training a single-agent RL model, training a multi-agent RL model, optimizing and comparing the implementations to find the best model, and using real world traffic data to validate the model as well as discussing the theoretical migration of the model to existing cities and networks worldwide. Python, with its extensive libraries, will be the primary language used. To manage the large volume of data, Google Collaboratory will be utilized, providing a cloud-based environment for efficient data analysis and integration with other Google services. Various tools such as git, GitHub, MsProject, LucidCharts, and GoogleDocs will support the project management needs. Computational times for training will also need to be reasonable to be able to complete the research project within the eight week duration. Microsoft Azure may need to be utilized as a virtual machine in order to decrease computational times and speed up the training process.

2.3 Assumptions

1) Availability and quality of driving datasets: The project assumes the availability of publicly accessible driving datasets containing reliable and comprehensive information about driving conditions, including congestion, traffic, accidents, and other relevant factors to validate the model.
2) Applicability of reinforced learning: The project assumes that reinforced learning can effectively provide optimal traffic light timing recommendations. It assumes that the model's performance can be improved over time through continuous learning and optimization.
3) The availability of the required technologies: The project assumes that the technologies currently available have the necessary features to simulate both single and multiple intersections in such a way that a multi-reinforced learning environment can interface with the simulation to train multi-agent reinforced learning models.
4) The ability to import real-world traffic data. One of the aims of the project is to integrate the models with real world data. This assumes that the simulators generated traffic can be replaced with a public dataset if approval is granted.

2.4 Constraints

1) Technical limitations: The project is subject to technical constraints, such as the computational resources required for model training and evaluation. The availability of computing power and the efficiency of the simulation software utilized, such as SUMO, may influence the scalability and speed of the project's implementation.
2) Physical implementation: Due to the project's scope and limitations, the practical execution of the simulated model is beyond its current focus.
3) Evaluation Metrics: Defining appropriate evaluation metrics to measure the effectiveness of the optimized light timings can be challenging.
4) Time constraints: The project's success depends on the timely approval of applications, datasets, and technologies. This includes ethical clearance and access to public driving datasets.

2.5. Risks and mitigations

Risks involved with this project and their mitigations are listed in the risk register in figure 1 of Appendix C.

3. PROJECT METHODOLOGY

The first phase of the methodology involves conducting thorough research on various aspects of the project. This includes researching different traffic simulators to identify the most suitable one for the investigation. Additionally, researching Multi-Agent Reinforcement Learning (MARL) Gyms and communication interfaces is necessary to understand their capabilities and determine the most appropriate options for the project. The research will also cover reinforced learning models, comparing existing solutions, and identifying the most effective techniques.

After the research phase, the project will move on to the implementation and evaluation stage. This involves setting up the required software and tools. Tutorials and documentation will be followed to ensure proper setup. The next step is to build, train, and evaluate different types of intersections using a single agent approach. This includes starting with a basic single agent intersection and gradually progressing to more complex single agent intersections. Multi-agent models will then attempt to be implemented to try and achieve better synchronization and coordination between multiple intersections.

Feasibility will be assessed using public datasets within the simulation, exploring methods to replace simulated traffic with real-world data if feasible. The learned models will be compared to traditional timings to measure their effectiveness. Additionally, strategies for migrating the learned models to

the real world will be investigated, considering the challenges and requirements for integrating the models into existing traffic networks.

The final phase of the methodology involves documenting the entire investigation project. A comprehensive report will be prepared, highlighting the research conducted, methodology employed, implementation details, evaluation results, and analysis. The report will also include discussions on the feasibility of migrating the models to real-world applications. Additionally, a presentation will be created to communicate the project findings and outcomes effectively.

The proposed methodology and division of tasks can be seen in table 1 of section 4.

4. PROJECT MANAGEMENT

4.1 Objectives

Six main objectives of this project have been identified and are described below:

1) Implement reinforced learning techniques to train machine learning models with the aim of optimizing traffic light timings
2) Thorough research conduction for project feasibility, planning, and implementation.
3) The solution must have the advantage of being more reliable and efficient over traditional solutions
4) Documentation in compliance with the Universities standards must be carried out through project reports.
5) Investigate the feasibility and challenges of migrating the developed model to existing cities and traffic networks worldwide, including a theoretical discussion of potential implementation strategies.
6) The completion of all tasks before Friday the 3rd of November

4.1. Project schedule/timeline

A number of milestones have been set out to encourage contributors to reach the different stages of the project. Milestones also aid in helping the contributors to see where they are in the progress of the project. The following list provides the milestones of this project :

- Completion of all research
- Complete environment installation and setup with tutorials
- Build, train, and evaluate a single agent intersection
- Build, train, and evaluate a multi agent combination of intersections
- Comparison of the learned models with traditional robot timings
- Replace SUMO's generated traffic with a public driving dataset
- Investigate strategies for exporting the learned models to the real world
- Completion of the final report

4.2. Work Breakdown Structure and Gantt Chart

The tasks required to finish the investigation are outlined in the work breakdown structure shown in figure 1 of appendix A. The Gantt chart in Appendix B's figure 1, illustrates the timeframes for each task and the overall timeline for completing the investigation project. To enhance the workflow and handle the workload effectively, bigger tasks are divided into smaller ones.

4.3. Documentation

Engineering notebooks will be kept by each member of the group to record activities and progress throughout the investigation project.

All documentation including reports, notebooks, meetings, drafts and applications must be in compliance with the schools Red and Blue book.

The final individual project reports will be completed by the 27th of October 2023 reflecting the work of each member. These reports will provide a holistic overview of what was done, how it was done, and an analysis of the results.

A group presentation must then be completed and submitted by the 3rd of November 2023 that is approximately 15 minutes in length, with each group member having approximately the same amount of time to talk.

4.4. Division of Work Between Group Members

The tasks that make up the investigation project were split between the two members as outlined in table 1 below.

Table 1: Division of Work Between the Group Members

Task	Group member	
	Bryce	**Michael**
Research traffic simulators		☒
Research MARL Gyms	☒	
Research communication interfaces	☒	
Research reinforced learning models		☒
Research and compare existing	☒	☒

solutions		
Software setup and tutorials	☒	☒
Build, train, and evaluate a basic single agent intersection	☒	☒
Build, train, and evaluate a complex single agent intersection	☒	☒
Build, train, and evaluate a multi-agent combination of intersections	☒	☒
Build, train, and evaluate a complex multi-agent combination of intersections	☒	☒
Verify the feasibility of using public datasets within the simulation		☒
Comparison of learned models to traditional timings	☒	☒
Investigate strategies for migrating the learned models to the real world	☒	
Analysis of results and performance	☒	☒
Improvements	☒	☒
Report writing	☒	☒
Presentation	☒	☒

4.5. Budget

Table 2 represents how the designated budget of R1800 will be allocated if necessary.

Table 2: Budget allocation

Item	Reason for requirement	Cost
Travel	In case the team requires physical meetings at an institution or venue outside of Wits where additional fuel expenses must be budgeted for	R400
Cloud based services	The use of a cloud based virtual machines for reduced compilation times of our Multi-agent reinforced learning models	R700
Project management tools	In order to develop and adhere to a project schedule the use of MsProject must be budgeted for. (Free trial periods)	R0
Research materials	For ebooks, textbooks, and other research papers that are neither free nor found in the Wits libraries	R600
Meetings and communication	Data and airtime when meetings and calls are required with team members or information sources	R100
Total		R1800

5. RESOURCES

The feasibility of this investigation project is dependent on three main technologies. The simulator which is responsible for simulating the intersections. A 'gym' for training reinforced learning models and an interface to facilitate any and all communication between the two.

All code, documents, and datasets will be source controlled using git and github. Both the training and simulation will make use of a cloud based virtual machine for improved performance and reduction in computation time. The entire project will be managed using project management tools such as MsProject.

Textbooks, Journals, and research papers will aid in the research and planning of the project. Tutorials and github repositories will aid in the setup and implementation stages. And lastly any advice and guidance from experts with machine learning knowledge and experience will be used throughout the project.

6. CONCLUSION

In conclusion, this investigation project aims to address traffic congestion in urban areas by leveraging machine learning techniques to optimize traffic light timings.

The project description, proposed implementation, objectives, milestones, work breakdown structure, budget, and risks have all been outlined in this report. By following the timeline in the Gantt chart, the investigation project will be completed before the deadline of the 3rd of November 2023.

REFERENCES

[1] M.s Netshisaulu, *How do urban cities manage road traffic and congestion*, Gauteng, pp.531-532, October 2021.

[2] SICE, "150 years of traffic lights", 06 February 2019. [Online]. Available: https://www.sice.com/en/news/150-years-of-traffic-lights. [Accessed 09 July 2023].

[3] C. Paduraru, M. Paduraru, & A. Stefanescu, "Traffic Light Control using Reinforcement Learning: A Survey and an Open Source Implementation", *8th International Conference on Vehicle Technology and Intelligent Transport Systems,* Bucharest, April 2022.

[4] Intelligent Transportation Systems, "Implementation Costs for Automated Red Light Camera Systems Range from $67,000 to $80,000 per Intersection", 30 September 2003.

[Online]. Available: https://www.itskrs.its.dot.gov/its/benecost.nsf/ID/2b209ad2c5ad2ab985256db10045892b. [Accessed 10 July 2023].

[5] H.R. Deekshetha, A.V.S. Madhav, & A. Tyagi, *Evolutionary Computing and Mobile Sustainable Networks*, ResearchGate, Chennai, pp.969-983, January 2022.

[6] M. Steingrover *et al,* "Reinforcement Learning of Traffic Light Controllers Adapting to Traffic Congestion", *BNAIC 2005 - Proceedings of the Seventeenth Belgium-Netherlands Conference on Artificial Intelligence,* Brussels, October 2005.

Appendix A

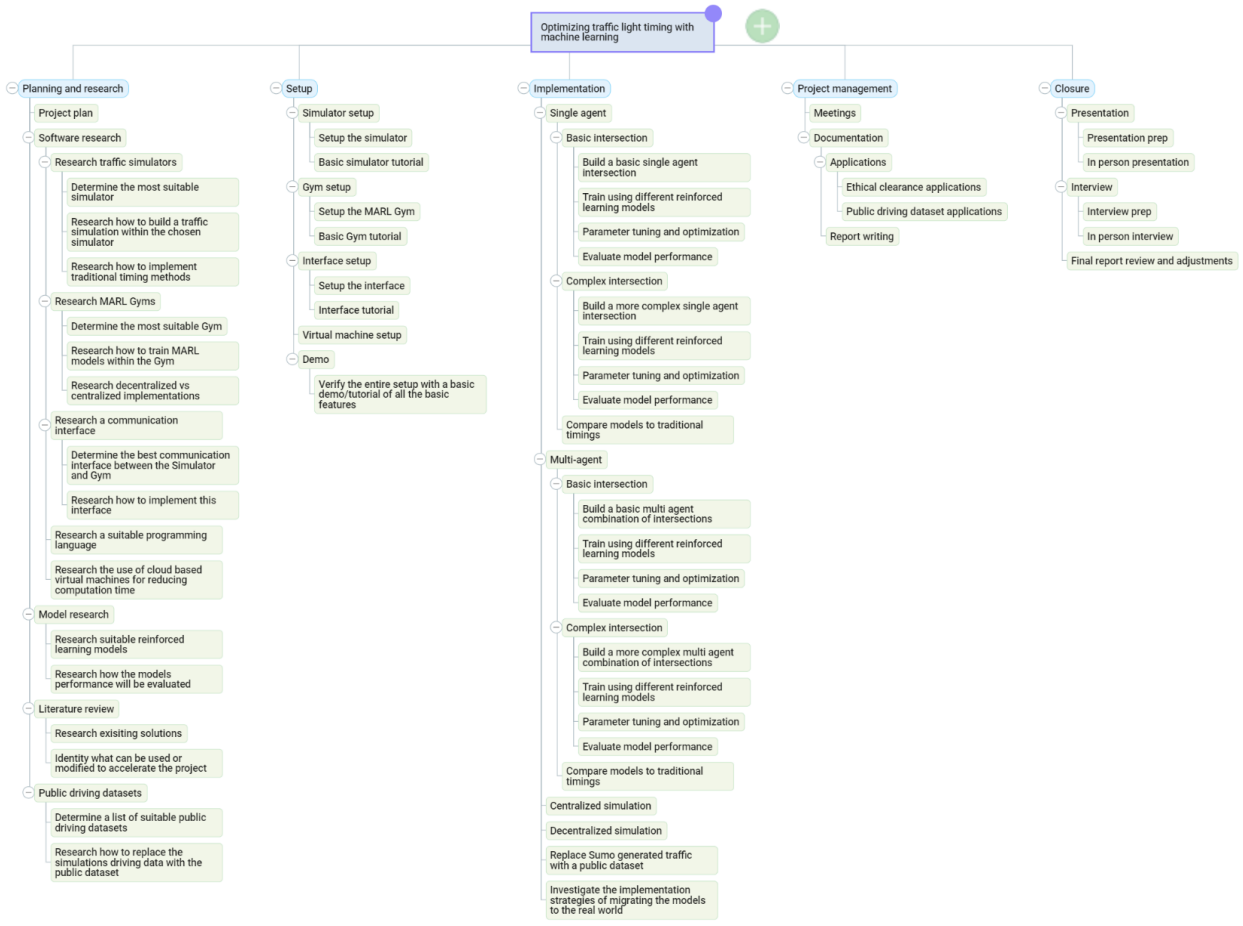


Figure 1: Work Breakdown Structure

Appendix B

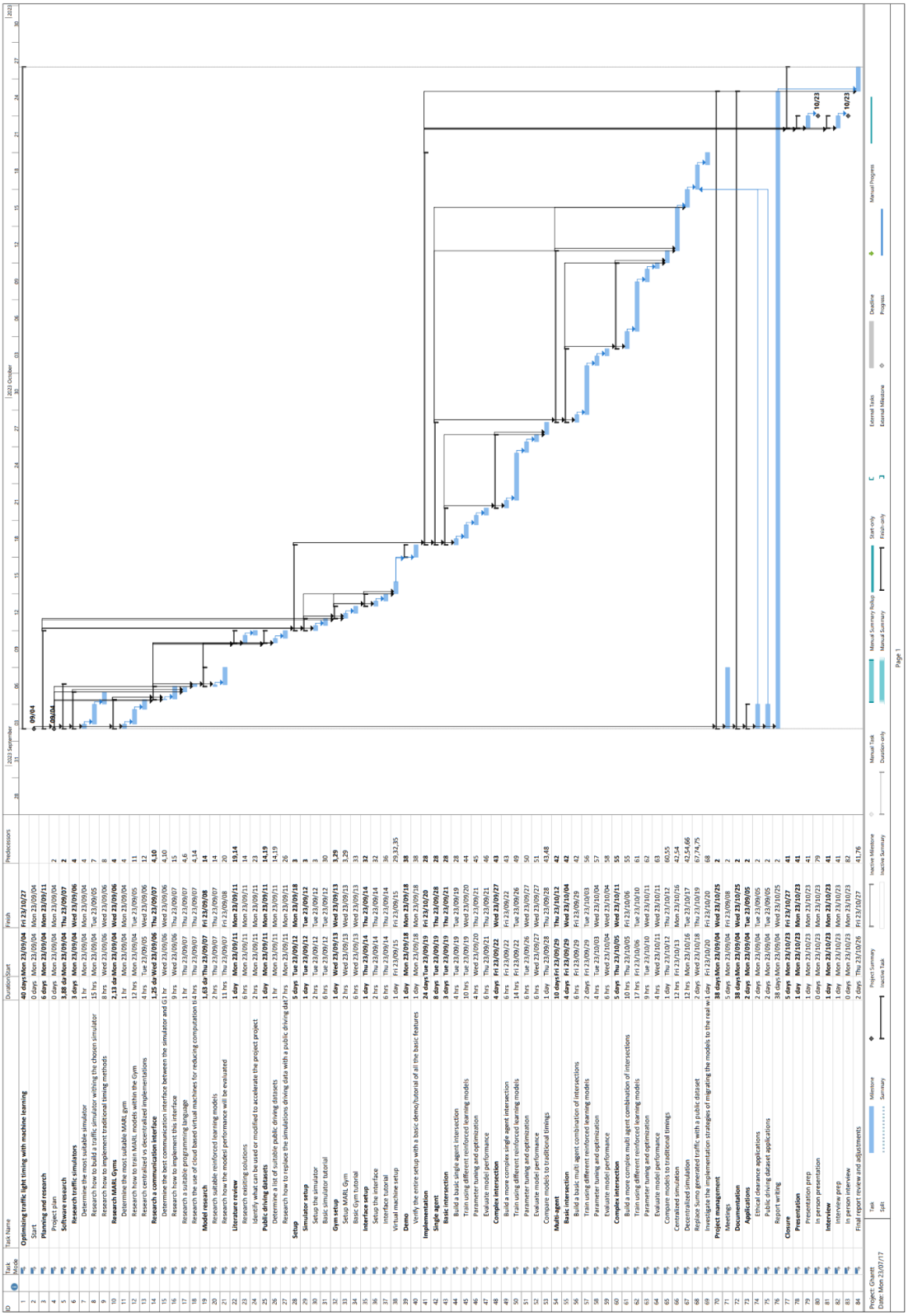


Figure 1: Gantt Chart

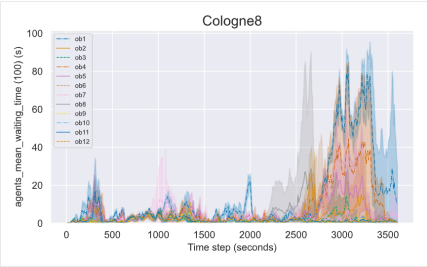
Appendix C

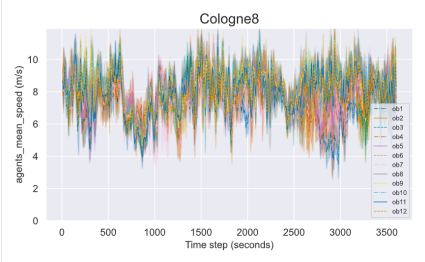
RISK REGISTER						Date			
Risk	**Causes (due to ...)**	**Probability**	**Impact**	**Risk Rating**	**Response**	**Actions**	**Start**	**End**	**Responsible**
There is a possibility that the start of the project is delayed	The ethical clearance takes longer than expected to be approved, being denied access to public driving datasets	1	2	2	Prevent	Apply as early as possible for ethical clearance and access to public driving datasets	28/06/2023	03/11/2023	Bryce Grahn & Michael Rolle
The inability to meet project deadlines	Long computational times, too large of a scope	1	3	3	Prevent	Limit the scope of the project, use Azure cloud services to reduce computational times if necessary	04/09/2023	03/11/2024	Bryce Grahn & Michael Rolle
The simulated model may not be acceptable to implement physically in the real world	Simulating all real world parameters such as pedestrians, potholes, accidents, speed limits, road closures etc. is too complex and computationally expensive	3	1	3	Accept	The physical implementation is out of scope for this project and so having a working model within the simulation will be acceptable	04/09/2023	03/11/2025	Bryce Grahn & Michael Rolle
Unseen traffic scenarios that are not included in the policy of the agent could cause unexpected behaviour	Limited training scenarios offered by the simulator. Unable to train the model on large amounts of scenarios due to long computational times	1	2	2	Prevent	Ensure to train the model on many different scenarios and have it constantly learn as well after implementation so that it can learn to handle new unseen scenarios	04/09/2023	03/11/2026	Bryce Grahn & Michael Rolle
The trained model performs poorly compared to traditional methods	Lack of experience and research papers	1	1	1	Accept	Reflect on the project and investigate what was done right or wrong and understand why it does not perform as expected	04/09/2023	03/11/2027	Bryce Grahn & Michael Rolle
Unable to meet all project goals such as a multi-agent model	Long computational times, too large of a scope, high complexity of multi-agent systems	2	2	4	Mitigate	Limit the scope to produce high quality results for what can be done instead of exploring too many options and designing poor solutions	04/09/2023	03/11/2028	Bryce Grahn & Michael Rolle

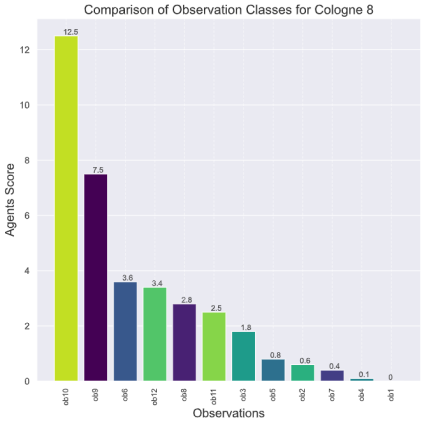
Figure 1: Risk Register

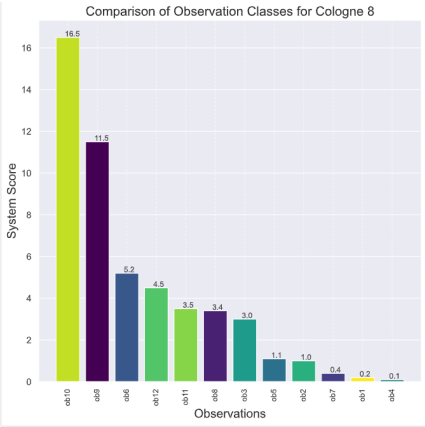
APPENDIX C: ADDITIONAL RESULTS

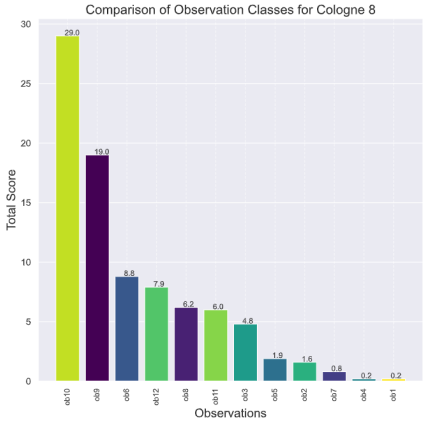
1. Observations

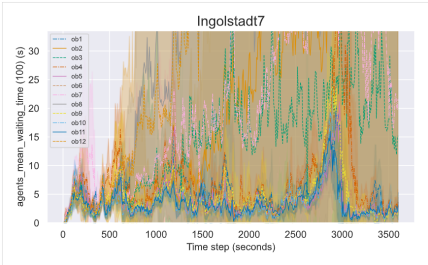


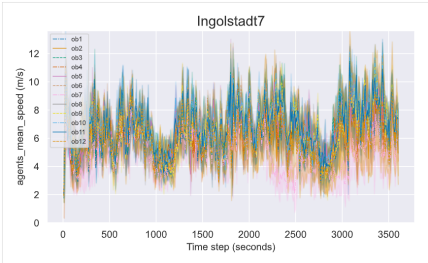


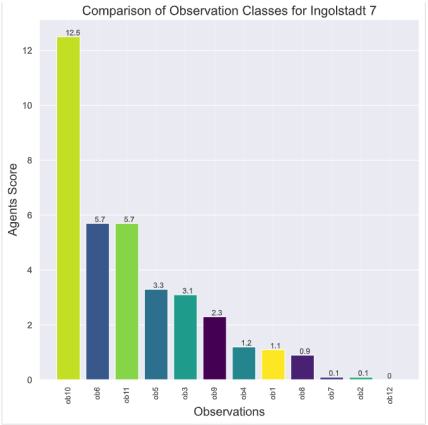


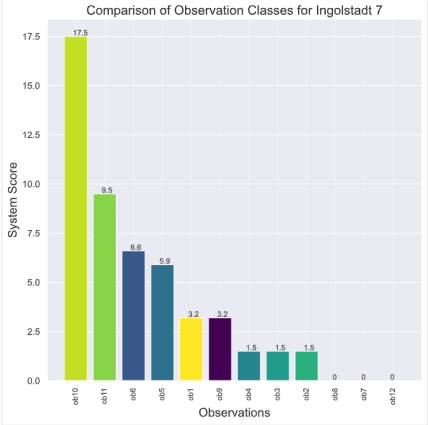


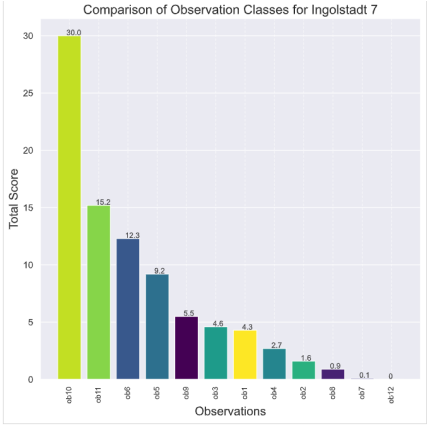




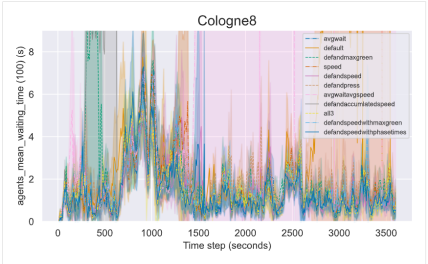


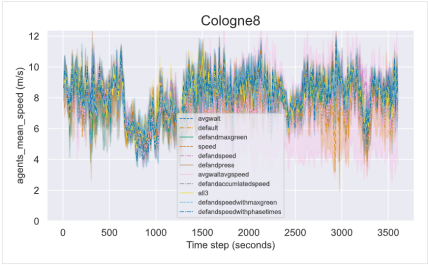


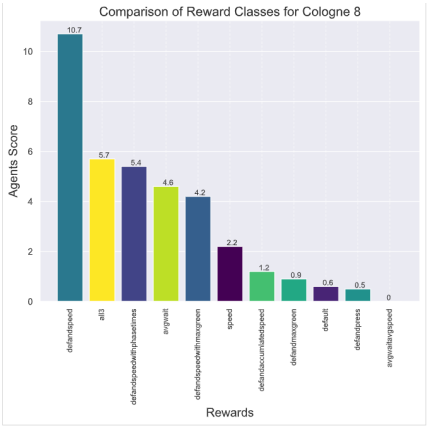


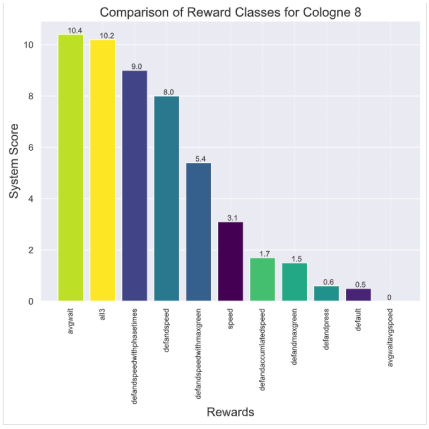


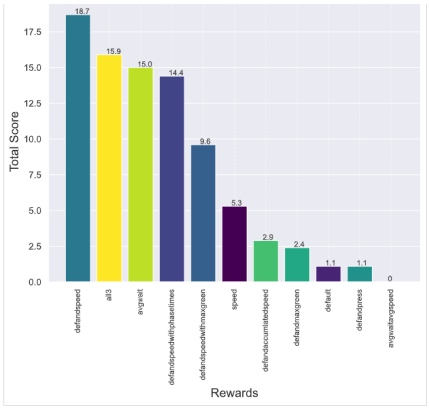
2. Rewards

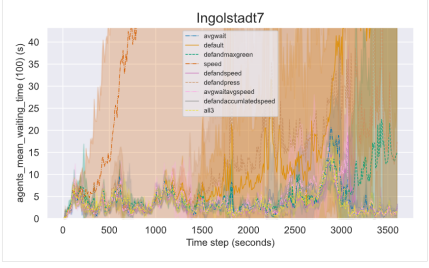


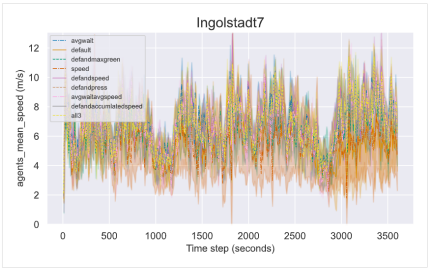


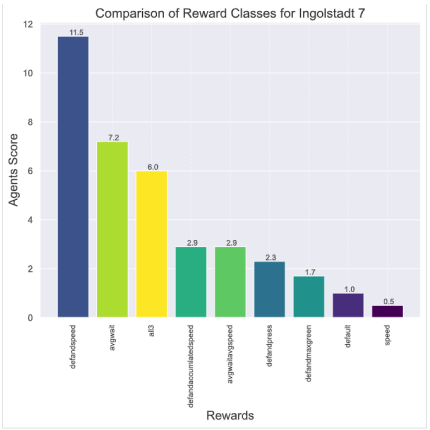

Comparison of Reward Classes for Cologne 8
Agents Score
Rewards
defandspeed
all3
defandspeedwithphasetimes
avgwait
defandspeedwithmaxgreen
speed
defandaccumulatedspeed
defandmaxgreen
default
defandpress
avgwaitavgspeed

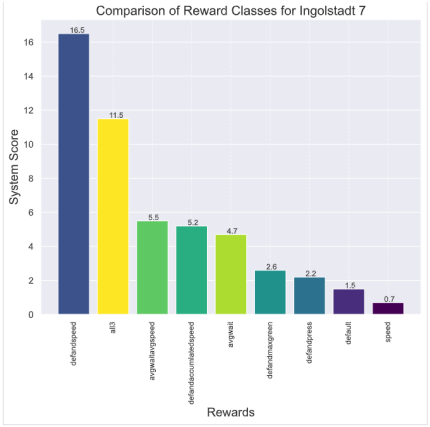

Comparison of Reward Classes for Cologne 8
System Score
Rewards
avgwait
all3
defandspeedwithphasetimes
defandspeed
defandspeedwithmaxgreen
speed
defandaccumulatedspeed
defandmaxgreen
defandpress
default
avgwaitavgspeed

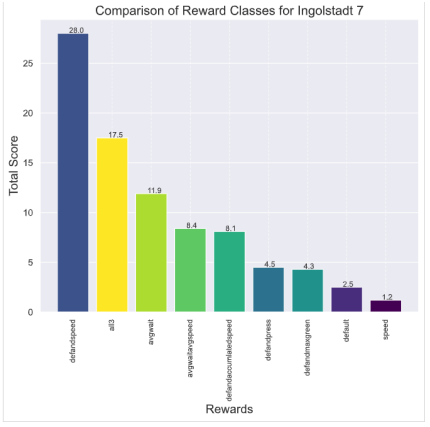

Total Score
Rewards
defandspeed
all3
avgwait
defandspeedwithphasetimes
defandspeedwithmaxgreen
speed
defandaccumulatedspeed
defandmaxgreen
default
defandpress
avgwaitavgspeed


Ingolstadt7
agents_mean_waiting_time (100) (s)
Time step (seconds)
avgwait
default
defandmaxgreen
speed
defandspeed
defandpress
avgwaitavgspeed
defandaccumulatedspeed
all3

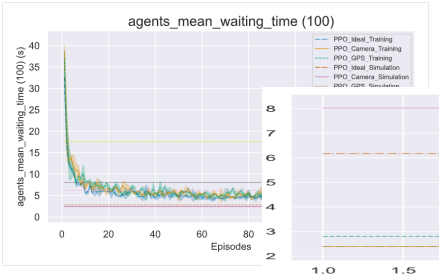

Ingolstadt7
agents_mean_speed (m/s)
Time step (seconds)
avgwait
default
defandmaxgreen
speed
defandspeed
defandpress
avgwaitavgspeed
defandaccumulatedspeed
all3

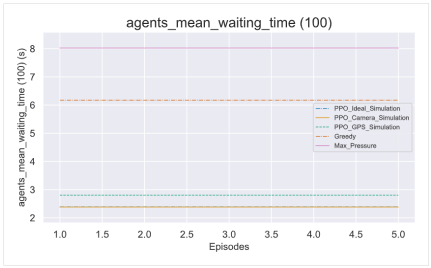

Comparison of Reward Classes for Ingolstadt 7
Agents Score
Rewards
defandspeed
avgwait
all3
defandaccumulatedspeed
avgwaitavgspeed
defandpress
defandmaxgreen
default
speed


Comparison of Reward Classes for Ingolstadt 7
System Score
Rewards
defandspeed
all3
avgwaitavgspeed
defandaccumulatedspeed
avgwait
defandmaxgreen
defandpress
default
speed

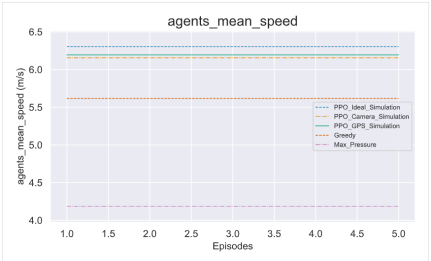

Comparison of Reward Classes for Ingolstadt 7
Total Score
Rewards
defandspeed
all3
avgwait
avgwaitavgspeed
defandaccumulatedspeed
defandpress
default
defandmaxgreen
speed

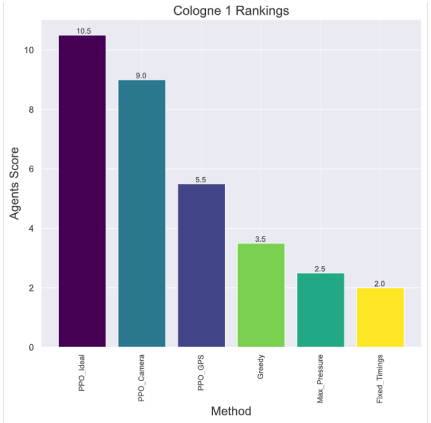
3. Cologne 1

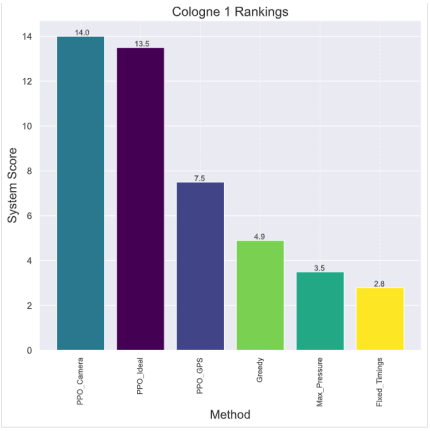

agents_mean_waiting_time (100)
agents_mean_waiting_time (100) (s)
Episodes
PPO_Ideal_Training
PPO_Camera_Training
PPO_GPS_Training
PPO_Ideal_Simulation
PPO_Camera_Simulation

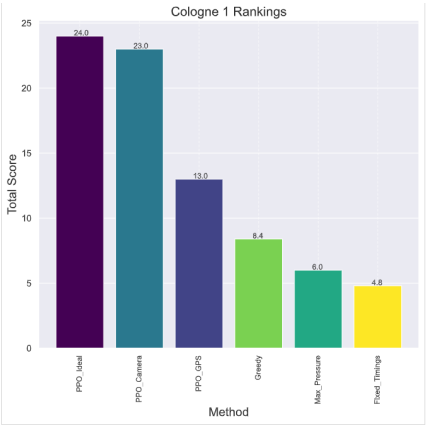

agents_mean_waiting_time (100)
agents_mean_waiting_time (100) (s)
Episodes
PPO_Ideal_Simulation
PPO_Camera_Simulation
PPO_GPS_Simulation
Greedy
Max_Pressure


agents_mean_speed
agents_mean_speed (m/s)
Episodes
PPO_Ideal_Training
PPO_Camera_Training
PPO_GPS_Training
PPO_Ideal_Simulation
PPO_Camera_Simulation
PPO_GPS_Simulation
Greedy
Max_Pressure
Fixed_Timings

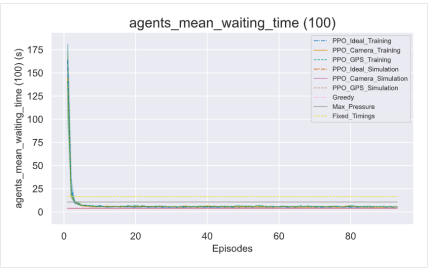

agents_mean_speed
agents_mean_speed (m/s)
Episodes
PPO_Ideal_Simulation
PPO_Camera_Simulation
PPO_GPS_Simulation
Greedy
Max_Pressure

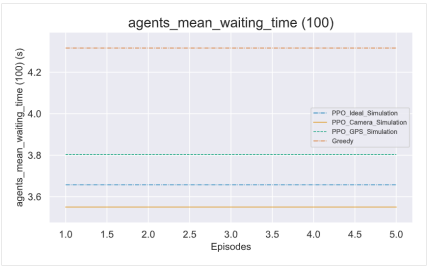

Cologne 1 Rankings
Agents Score
Method
PPO_Ideal
PPO_Camera
PPO_GPS
Greedy
Max_Pressure
Fixed_Timings

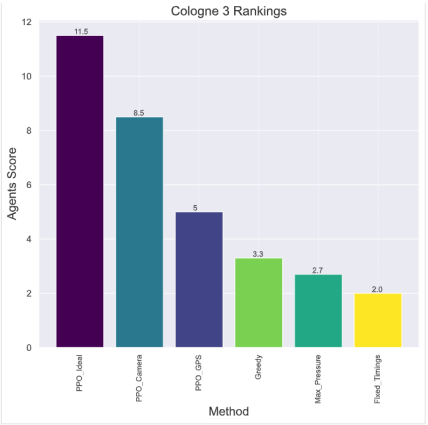


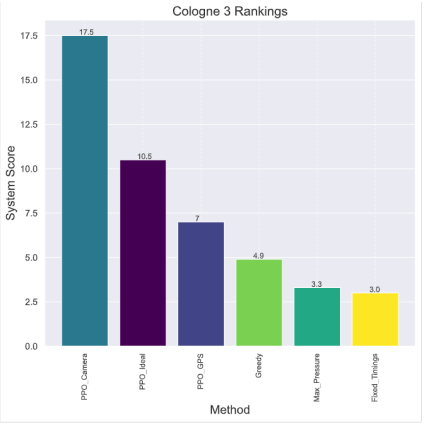


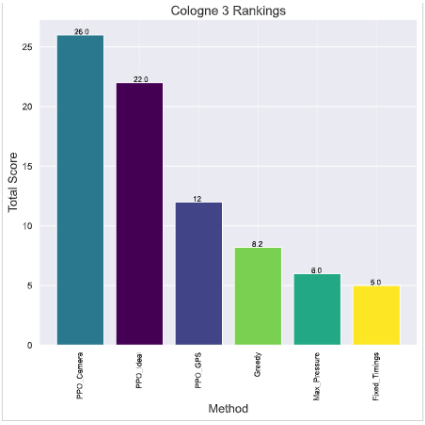
4. Cologne 3



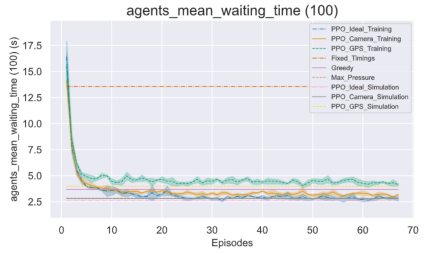


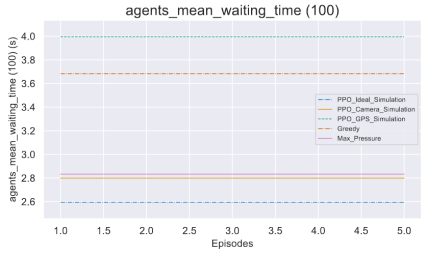


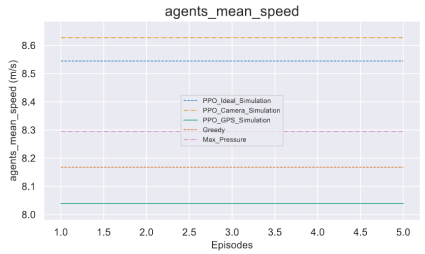


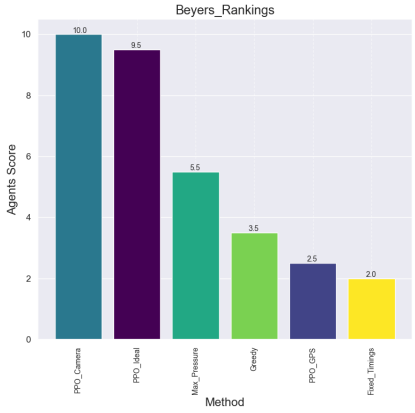


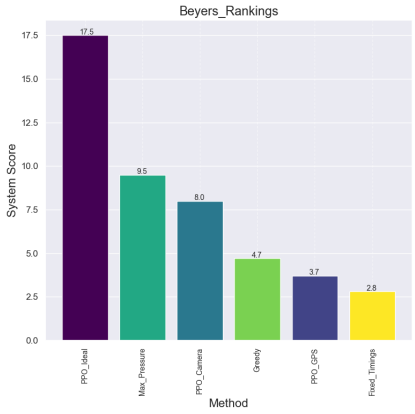
5. Beyers

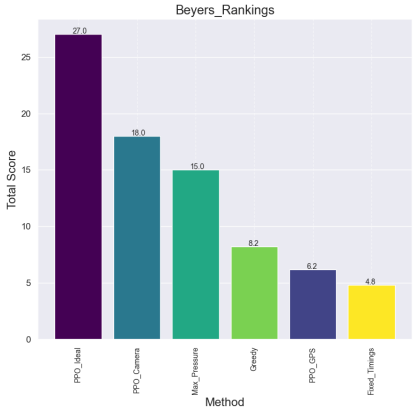




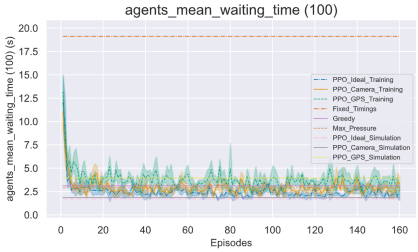


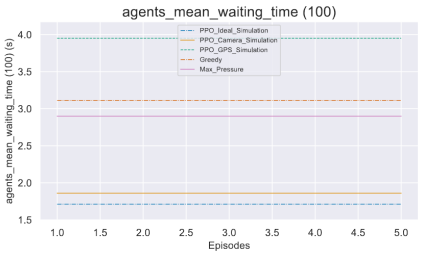


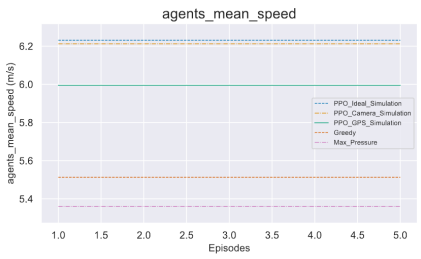


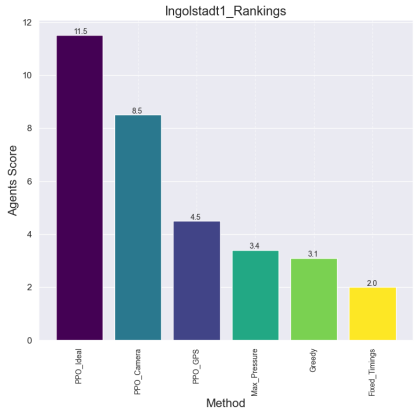


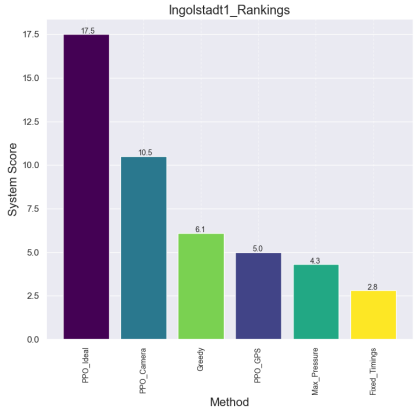
6. Ingolstadt 1

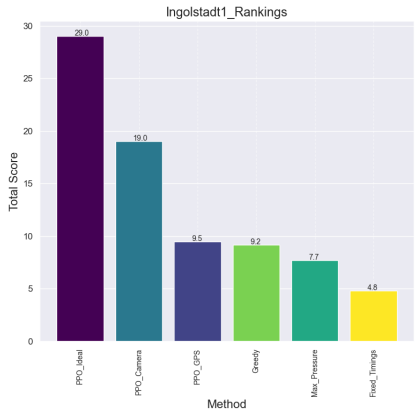




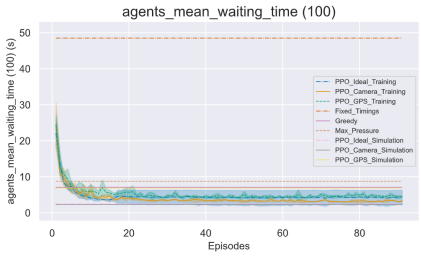


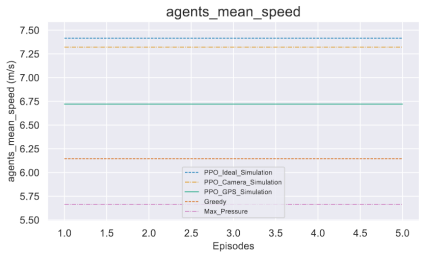


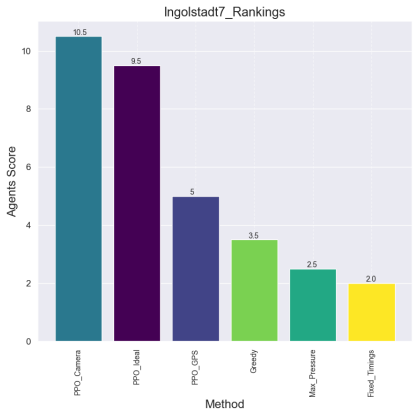


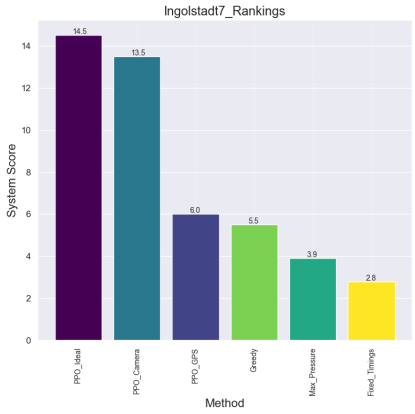


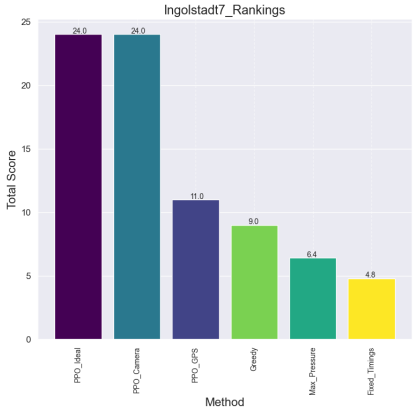
7. Ingolstadt 7



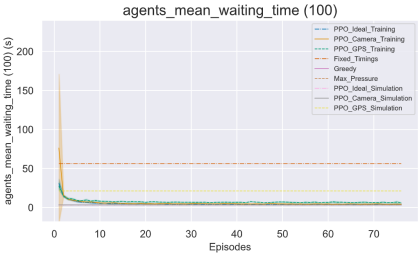


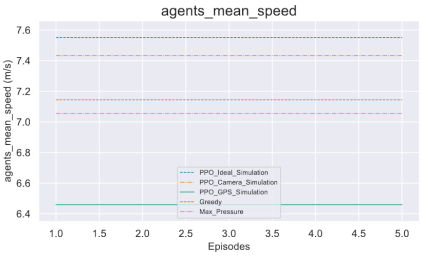


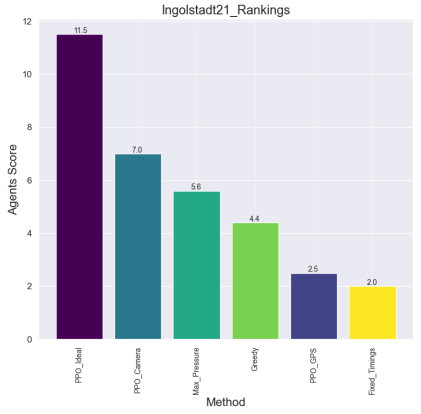


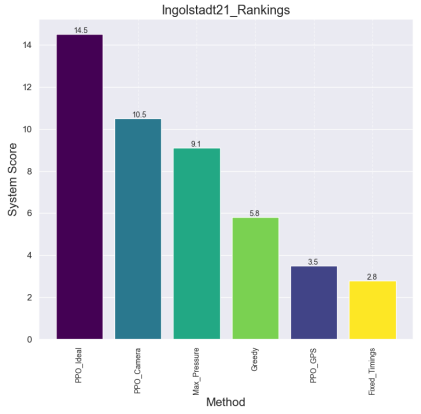


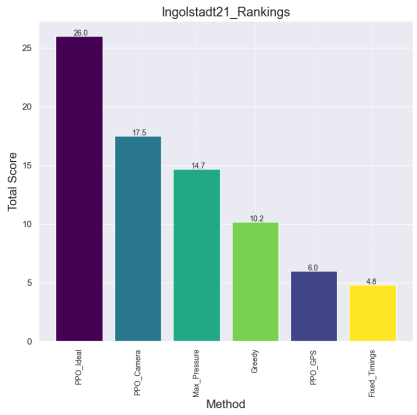
8. Ingolstadt 21











9. Hyper- parameter tuning:

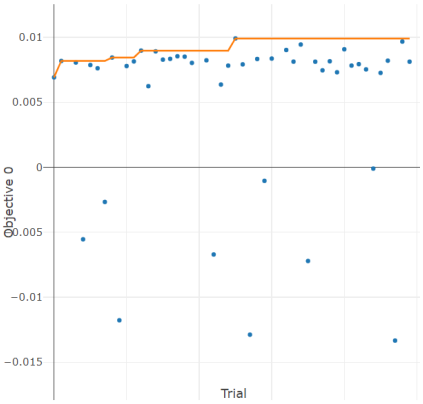
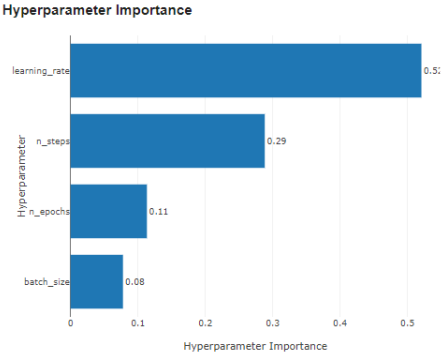


Figure 1: Trial history



10. Training improvements

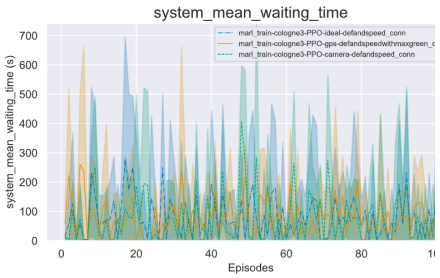


Figure 3: Error prone training

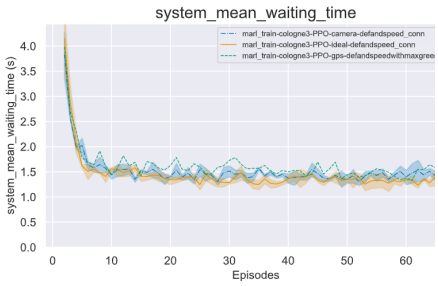


Figure 4: Errorless training

9. Overall comparison

Table 1: Agent Rankings:

Map	1st	2nd	3rd	4th	5th	6th
Beyers	Cam	Ideal	Max P.	Greedy	GPS	Fixed
Ing1	Ideal	Cam	GPS	Max P.	Greedy	Fixed
Ing7	Cam	Ideal	GPS	Greedy	Max P.	Fixed
Ing21	Ideal	Cam	Max P.	Greedy	GPS	Fixed
Col1	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Col3	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Col8	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Final	**Ideal**	**Cam**	**GPS**	**Greedy**	**Max P.**	**Fixed**

Table 2: System Rankings:

Map	1st	2nd	3rd	4th	5th	6th
Beyers	Ideal	Max P.	Cam	Greedy	GPS	Fixed
Ing1	Ideal	Cam	Greedy	GPS	Max P.	Fixed
Ing7	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Ing21	Ideal	Cam	Max P.	Greedy	GPS	Fixed
Col1	Cam	Ideal	GPS	Greedy	Max P.	Fixed
Col3	Cam	Ideal	GPS	Greedy	Max P.	Fixed
Col8	Ideal	Cam	GPS	Max P.	Greedy	Fixed
Final	**Ideal**	**Cam**	**GPS**	**Greedy**	**Max P.**	**Fixed**

Table 3: Total Rankings:

Map	1st	2nd	3rd	4th	5th	6th
Beyers	Ideal	Cam	Max P.	Greedy	GPS	Fixed
Ing1	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Ing7	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Ing21	Ideal	Cam	Max P.	Greedy	GPS	Fixed
Col1	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Col3	Cam	Ideal	GPS	Greedy	Max P.	Fixed
Col8	Ideal	Cam	GPS	Greedy	Max P.	Fixed
Final	**Ideal**	**Cam**	**GPS**	**Greedy**	**Max P.**	**Fixed**

APPENDIX D: ADDITIONAL DIAGRAMS AND TABLES

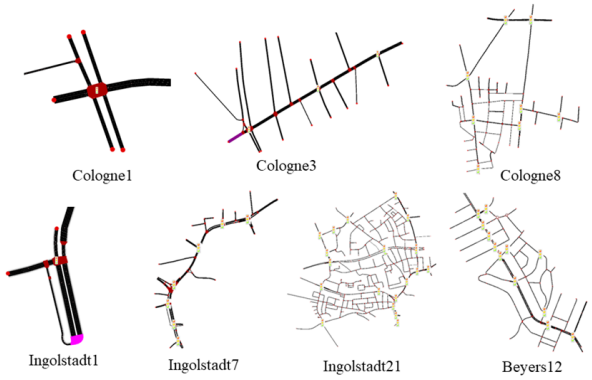


Figure 1: Sub regions of real world maps

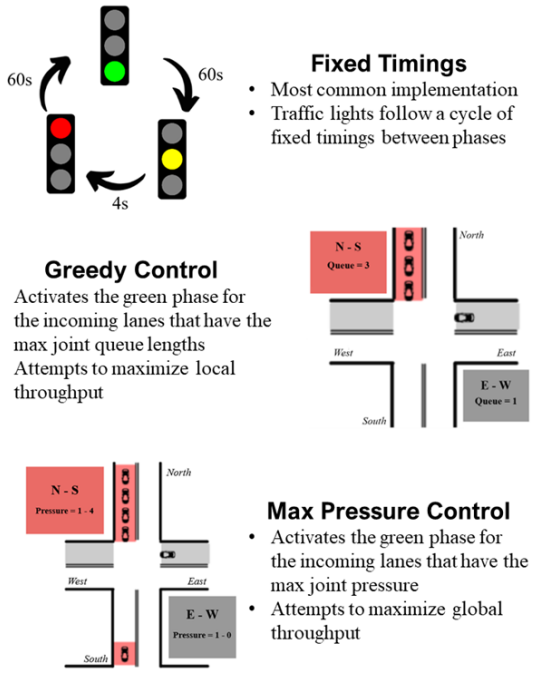


Figure 2: Fixed, greedy, and max pressure illustration

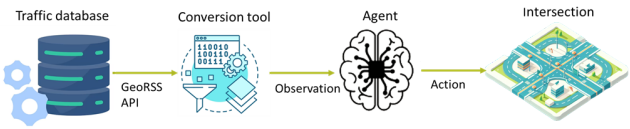


Figure 3: An illustration representing the interface between waze's public dataset and an intersection

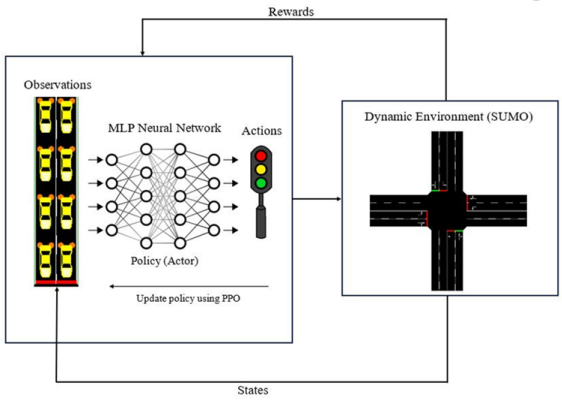


Figure 4: Reinforcement learning loop

Category	Option		
Type	Single-agent	Multi-agent	
		✓	
Simulator	Custom	Matlab	SUMO
			✓
Training Gym	Custom	OpenAi gymnasium	PettingZoo
			✓
Approach	Centralized	Hierarchical	Decentralized
			✓
Model	MADDPG	DQN	PPO
			✓
Observation	Fixed	Camera	GPS
			✓
Interface	Custom	Sumo	Sumo-rl
			✓
PPO toolkits	CleanRL	RLIB	SD3
			✓
Environment model	AEC	Parallel	
		✓	

Table 1: A table representing important design decisions

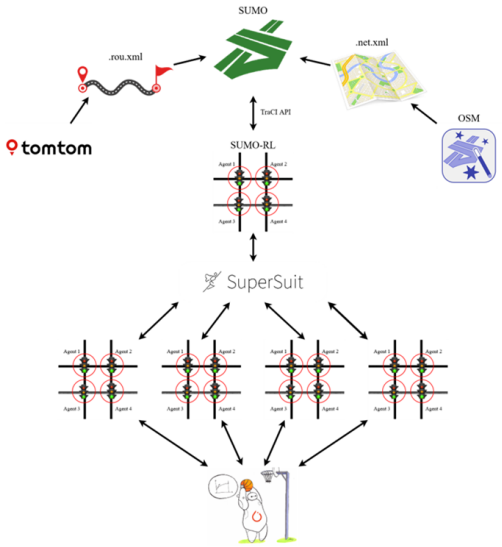


Figure 5: The relationships between the various technologies used

Yellow Interval Duration

A commonly used equation for calculating the duration of the yellow interval is that proposed by the Institute of Transportation Engineers' (ITE) Technical Committee 4A-16 as a recommended practice (*6*). This equation is:

$$Y(v) = T_{pr} + \frac{V_a}{2d_r + 2gG_r}$$

where:
$Y(v)$ = yellow interval evaluated at speed $V_a = v$, s;
d_r = deceleration rate, use 10 ft/s^2;
g = gravitational acceleration, use 32.2 ft/s^2;
G_r = approach grade, ft/ft;
T_{pr} = driver perception-reaction time, use 1.0 s; and
V_a = speed of vehicle approaching the intersection (typically the 85th percentile speed), ft/s.

Figure 6: Yellow light duration

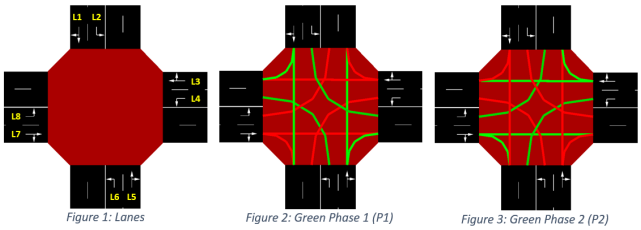


Figure 7: 8-lane intersection

APPENDIX E: COHORT 5 MEETING MINUTES - [EIE Investigation].

1. WEEK 2

Location: 2nd floor Chamber of Mines, EIE Seminar Room.

Date: 15th September 2023.

Time: 09:00 am

Adjournment: The meeting was concluded at 10:08 a.m. The next meeting is to take place on the 22nd of September 2023, at the same location, led by group 38.

Submitted and approved by: Themba Mtshelwane (Week 2 Secretary) and Sifiso Mzobe (Week 2 Chair).

Attendance:

- Groups: 23G52, 23G38, 23G06, 23G29, 23G24, and 23G34.
- Supervisors: Estelle Trengove and Martin Bekker.

Agenda Items:

1. Each group will introduce themselves.
2. Briefly explain what their project is about.
3. Did the group achieve what they outlined they would get done for week 1 in their project plan? If not, what do they think the problem was?
4. Did the group achieve what they outlined they would get done for week 2? If not, what do they think the problem was?
5. Get the supervisor's feedback/advice on any problems the group is currently going through.

Opening

- Formal introduction and welcoming by Sifiso Mzobe at 9:02 a.m.

Discussion

- **Group 06** introduced their project, the Chichen Itza Stairs. The project is about modelling the phenomenon that occurs at the Temple of Kukulkan, i.e., when an observer stands in front of the temple and produces a clap and echo that sounds like a quetzal bird. During week one, group 06 obtained six clap-echo data samples from YouTube using YouTube-dl; those samples were processed using FFMPEG and Audacity software. For week two, group 06 performed both time and frequency analysis on the samples and determined features of interest, for example, a signal's zero-crossing rate. Group 06 also noted a delay in their equipment procurement and testing. It was suggested that the group collect more data samples to improve the analyzing process.

- **Group 52** introduced themselves and their project, creating an app that will use machine learning to convert black and white images to colour. Their progress includes collecting about 5000 image samples, designing an app interface using Figma. They used CNN and VGG16 machine learning algorithms. They expanded on the VGG16 model they used its feature extraction over classification functionality. Their current model has an accuracy of about 20% and they are planning to fine tune it. A few suggestions were given by the supervisors. Firstly, to not build the app from scratch, instead they could implement the model on a computer, secondly use one type of image, either pictures of landscapes or faces. If they used faces, they should consider the biases their model could have.

- **Group 38** introduced their project, creating a model that will transcribe audio that is in isiZulu. For week one they focused on identifying sources of isiZulu samples, which they acquired from the Department of Arts and Culture and the Department of Technology. Group 38 faced a problem with the data samples they acquired, realizing that some of the samples were incorrect. During week two the group focused on how to build and train a language model.

- **Group 29** followed with introducing their project of determining driving culture through analyzing publicly available live-feeds of traffic. Features that they consider fitting to determine driving culture includes driving speed, lane changes and traffic law obedience. During week one group 29 focused on collecting data. They noted that they have access to traffic live-feeds from three different countries. They noted that they were exploring algorithms such as Yolo and SSD for vehicle detection. For week two, group 29 analyzed driving culture using Yolo, which analyzed features such as distance, speed and car type. A few questions were asked to the group. The live feeds are from traffic lights and in private properties. The group prefers using Yolo as it has classes of objects pre-trained on the features the group wants to focus on, i.e., driving speed, etc. The group also noted that they had problems with their devices' processing power. Two solutions were suggested firstly to use the available Cluster and secondly instead of using a live-feed they could analyze a short, recorded video of traffic.

- **Group 24** introduced their project which is about simulating agents and how those agents interact with each other, their environment, and their likelihood of rebelling. For their proof-of-concept group 24 simulated an environment where an agent's wealth affected their happiness and thus their likelihood to rebel. For week one the group set up their IDEs and repositories; and tried to understand and implement agent-based modelling through Mesa. During week two group 24 considered creating their own model and using an agent's Human Development Index as a feature to measure their likelihood to rebel. The group noted that they faced problems with computer processing power and were advised to use the available Cluster. The group also noted that their simulations always ended with agents rebelling due to the conditions they set of how some agents acquired or more accurately had no means to acquired happiness. In the future group 24 plans to add more features, - for example, the GINI coefficient – to measure likelihood of rebellion and restrict the agent's movements. It was suggested that the group implement neighborhoods for the agents and consider using economic concepts as they might have more information about how factors such as wealth impact people. It was noted that this project does not necessarily have to pertain to happiness, it could also be used to track health.

- **Group 34** introduced themselves and their project. Their project is about analyzing social media, specifically Twitter now X, to see how certain events affect social media behavior/responses. The group had problems with accessing Twitter's data so for week one they researched ways to obtain open-source data and found resources such as GDELT and Kaggle. During week two the group planned to extract information of three main events from the open-source data they have obtained. Mainly get data about the USA election during 2020, the social unrest in South Africa during July 2021 and lastly the Turkey-Syria Earthquakes in early 2023. They group also plans to compare traditional media response versus Twitter's social response. They noted that they currently have about 200GB worth of data available and need to process it. It was suggested that they used the available Cluster to quickly and efficiently extract and process the required data.

- General notes were given by both supervisors, that students should always note their assumptions, have well commented code and for students to document their project's progress in their notebooks and that for open day the students must at least have something that works to present.

2. WEEK 3

Location: 2nd floor Chamber of Mines, EIE Seminar Room.

Date: 22nd September 2023.

Time: 09:00 am

Adjournment: The meeting concluded at 10:00 a.m. The next meeting is to take place on the 29th of September 2023, at the same location, led by group 52.

Submitted and approved by: Mhlengeni Miya (Week 3 Secretary) and Lethiwe Mchunu (Week 3 Chair).

Attendance:

- Groups: 23G52, 23G38, 23G06, 23G29, 23G24, and 23G34.

- Supervisors: Estelle Trengove and Martin Bekker.

Agenda Items:

1. What did each group achieve for week 3?
2. Get the supervisor's feedback/advice on any problems the group is currently going through.

Opening

- Formal introduction and welcoming by Lethiwe Mchunu at 9:00 a.m.

Discussion

- **Group 38** During week three, group 38 finished the Isizulu model that will be used for word and character accuracy of the mozilla pretrained output. Group 38 Obtained more IsiZulu data and did data processing to meet data training for the mozilla pretrained model. For the fourth week the plan is to train the mozilla even more for better results in Isizulu.

- **Group 06** The project is about modelling the phenomenon that occurs at the Temple of Kukulkan, i.e., when an observer stands in front of the temple and produces a clap and echo that sounds like a quetzal bird. During week three, group 06 had planned to produce the Deconvolution method. They obtained about 101 samples from 25 different videos which was quite a long process,They were in the process of finding impulse responses for the samples but they were not sure on whether to find the impulses in time or frequency domain. The Supervisors suggested that they consult with a signals and systems lecture to gain more insight on that.

- **Group 52** was focusing on finding and training new models. They found landscape models that are mainly dealing with images of Forests, buildings and trees. When they trained the model they realized that it performs better than before, it colorizes the images a lot better than before but still not great. One of the issues they are facing is the time it takes to train the model which is an average of four to six hours. They had a question about conducting a test to gauge how realistic the image colours are, would it be trying to determine how accurate it was by just looking at the images and analysing the output obtained. The supervisors suggested that the get a model they can train with black and white images as input coloured images as output and then compare the two outputs

- **Group 29** During week three group 29 realized that their initial idea to analyze vehicles at intersections and highways was not going to work for intersections. Intersections does not provide adequate information about the driver's behavior, so the focus now is to only study the cars while on the highway. They managed to create a demo for their final product, but the code still contained a lot of errors so they decided to split the work so that each member focuses on the different

algorithms for their product. one member is focusing on doing the line change in following distance. The algorithm used to implement this is similar to the algorithm that was used to determine the distance between two people during covid19. The issue was that it identified all the objects that were in the frame even if they are not vehicles and makes the data required unreliable. This member mentioned that he is tracing the lines on the road to mark the vehicle position and he noted that this method takes up a lot of memory to process, the plan was to then optimise it by using regions instead of lines. The other member reported that he worked on the detection and tracking of vehicles where he used pixels to estimate the number of vehicles that pass a specific point at a given time. The line was drawn on the road to use as a point of reference to capture the vehicle. He also worked on speed estimation where he used two lines as the starting and final point then calculated the number of pixels in between the lines, the divide it by time taken to cross the two lines. In the process he got confused on how he is going to capture the time a starting and final point. Dr Bekker suggested that he looks into the position of the camera and calculate the distance between the camera and the lines and use the pythagoras approach.

- **Group 24** Focused on model implementation which involved choosing a coding language for the project, setting up a coding environment, necessary libraries and frameworks. They aimed to implement agents classes, find attributes and behaviors and how they interact with one another. For the fourth week they said they would be working towards implementing the environment which is to create a structure to represent environments and properties. They would also like to integrate agents interaction and code mechanism that control those interactions within the environment.

- **Group 34** Focused on expanding the data set on each event they are going to analyze, they looked at hashtags, keywords and location they are from, to find the data set between the conventional media and twitter. They plotted the location of the tweets on the map plan and also did the same for the conventional media. They said they had discussed with their supervisor that they should look at frameworks to look for specific patterns like weekly, daily, and seasonal patterns for analysis. As well as regression analysis to draw models. The challenge they faced was that they weren't sure if the datasets are good enough to validate the study of social media and conventional media impact, for example if an influence were tweet about a matter that would skew the curve and the conventional media curve could remain steady because media will not publish as much as tweeps would tweet.

- The supervisors requested that from week three meeting onwards the groups should bring their laptops with them to demonstrate their work so far.

3. WEEK 4

Location: 2nd floor Chamber of Mines, EIE Seminar Room.

Date: 29th September 2023.

Time: 09:00 am

Adjournment: The meeting concluded at 10:30 a.m. The next meeting is to take place on the 06th of October 2023, at the same location, led by group 34.

Submitted and approved by: Karabo Kgatla (Week 4 Secretary) and Janet Gapare (Week 4 Chair).

Attendance:

- Groups: 23G52, 23G38, 23G06, 23G29, 23G24, and 23G34.

- Supervisors: Estelle Trengove

- Apologies: 23G38 – Mhlengeni Miya

Agenda Items:

1. Did each group achieve their objectives for week 4? If not, what do they think went wrong.
2. Demonstrate what each group has worked on thus far.
3. Get the supervisor's feedback/advice on any problems the group is currently going through.

Opening

- Formal introduction and welcoming by Janet Gapare at 9:00 a.m.

Discussion

- **Group 38:** The plan was to combine the models for text and speech and calculate the accuracy from the output. The group used Google Collab, where they faced challenges running Google Speech due to requiring a Linux system. However, technical issues were observed with using the Linux subsystem. The model Language model couldn't be run on Linux. The team mentioned the libraries were not download or installed correctly and a suggestion was made to use pythons' pip to install libraries instead or searching on Stackoverflow. The group was therefore not able to demonstrate due to technical issues hindering inability to mount their data via Google drive or Google collab. They plan to create a local script to process the data locally. Other groups suggested they try running their code on Docker containers. Another suggestion was made to use GitHub instead of Google Colab.

- **Group 34:** The group demonstrated a graph showing tweets per day and news articles being published for the Social Unrest, and US elections. It was observed the news articles peaked after Twitter events, possibly due to the process news outlets take in to publish an article. The group also noted that the number of news articles published decayed much slower than Twitter events. The supervisor said that could be due to journalists dragging out stories and trying to conduct more interviews whereas Twitter trends don't last as long due to Twitter users' short attention span on a topic. The group plans to investigate the patterns more. This may possibly lead them to know by looking at a particular graph what the event that led to that graph was.

- **Group 24:** For the week the team limited the radius of the agents by bounding them to a radius, and combined safety done for the previous week. The team started creating their own custom visualization, which is currently still needs improvement. They plan on focusing on visualization, where they will be looking detecting jumping data and smoothing the process. They demonstrated a graph showing agents beginning with happiness at 0, which changes over time. The Gini was demonstrated using a happy coefficient. They also focused on showing the safety vs happiness, where the safety was constant. The results showed correlation between happiness and safety, where agents became happier with higher safety. They also demonstrated happiness for a specific agent. In their graphs, different colors demonstrated various levels of happiness from red, being the least happy agent to green being the happier agent as agents interact with one another. The group would like to show the above graph on open-day, as well as add more parameters they can switch through for visualization. The group also plans to perform stress tests on the model. The group hopes to get the cluster running by next week.

- **Group 29:** The group planned on fixing speed, following distance and detection of lane changes, which was archived. The group managed to plot their data in real-time, detection of traffic log violation – which included detecting speed limit violation, whether headlights are on/off, traffic light violation, and lane change without signaling. As suggested in previous week the group managed to change the camera perspective for calculating the car speed by changing how they calculated distance. The group concept known as Perspective transformation, which allowed the group to convert point of view of a camera to a bird-eyesight view to calculate the distance. The group calculated pixels/meter and converted it to meters. The group obtained inaccurate results of speed before due to how time was obtained. This was due to the processing time changing for various models used, as a solution the team subtracted the processing time from the actual total time. The team demonstrated their model by running the code locally, showing real-time plotting showing data such as speed. The limitation they faced initially was on highways, due to lack of traffic lights, thus the group switched to observing intersections instead. The group plans to show real-time plot while running the video in real-time. To detect lane changes, the group used the regions as lanes, and used the ray-casting method to show the line change. A question was asked about the distance between cars which they answered they used the perspective transformation to get a better calculation of the speed.

- **Group 06:** The group demonstrated a sample of their dataset indicating clapping sounds. The group used various models to achieve their goal. The first model used a proof-of-concept that if a clap and a random echo, it results in an echo of the clap. Initially they tested using a clap they recorded and convolved with Transfer Function to get the clap-echo. This method was abondend after analysing the Impulse Response. The second model used was using the deconvolution method. The model involved averaging the frequency domain and time domain. The impulse response for echo 3 was determined to be better. After searching through 101 samples, 3 of the impulse responses were closer to that found for echo 3. The team plans to add the 3 samples and average them. RMSE was not the best method for comparing the results. The team aims to use their microphone for recording participants' claps and demonstrating their model. A question was asked about how samples were obtained; the team responded that 25 samples were obtained from You-Tube. The group was also asked about issues in comparing sound echoes. The team mentioned issues may arise from not knowing the distance participants' were from the microphone in the sample data. A suggestion was made that the team could scale the predicted outputs to be the same. The supervisor also suggested the team slows down the recording of the clap and result for open day.

- **Group 52:** The objective of the team was to perform more training for their model. The group mentioned and demonstrated they initially trained their model using 1000 samples which resulted in poor images. After changing IDE from Jupiter to Spyder the results did become better although still good. The dataset was increased to 2000, which resulted in much better colorized image. The group also compared original-colored images and predicted images for which the results were better. After attempting to increase the sample space to 4000, their devices crashed. The plan is to try using a different device which may have better specs and can handle the increase in dataset. It was observed that the original-colored image was enhanced possibly using photoshop, and that the model actually predicted colors which were much closer to reality than the original-colored images. A question was asked why the image ratio is different. A response was given that the required image input needed to be the same for training the model as well as input to predict. As a result, the images were reshaped in the model. A suggestion was made that they can manually crop the image before being used as input.

- The supervisor said they will check with the course coordinator regarding details for Open day. However, the groups should make their open-day posters with minimal texts, possibly using bullet points, more images and easier to understand for the general public.

4. WEEK 5

Location: 2nd floor Chamber of Mines, EIE Seminar Room.

Date: 6th October 2023.

Time: 09:00 am

Adjournment: The meeting concluded at 10:30 a.m. The next meeting is to take place on the 13th of October 2023, at the same location.

Submitted and approved by: Devlan Mckenzie (Week 5 Secretary) and Zhuofeng Wu (Week 5 Chair).

Attendance:

- Groups: 23G52, 23G38, 23G06, 23G29, 23G24, and 23G34.

- Supervisors: Estelle Trengove.

Agenda Items:

1. What was your plan for this week, and did you achieve it? If not, what do you think was the problem?
2. Get the supervisor's feedback/advice on any problems the group is currently going through.
3. Demo what you've worked this week or anything that is new since last week.

Opening

- Formal introduction and welcoming by Zhuofeng Wu at 9:00 a.m.

Discussion

- **Group 24**: Greetings and project introduction. The group spoke about how they worked towards refining UI and interaction between the viewer and the application. They then went on to describe how they went about researching future indexes and how they ran light vs heavy simulations. They then went on to describe how they implemented contagion using riot threshold values and how they plan on implementing various initial distributions of wealth to run and evaluate the model. This was followed by general questions about the report and what is expected from it. The group demonstrated what visual aspects they have been working on for the week as well as a new index.

- **Group 52**: Greetings and project introduction. The group spoke about how they implemented structural similarity indexes for model testing and evaluation. This uses various elements such as brightness and contrast, among others, to evaluate how accurate the model is. This was demonstrated and they spoke of how they achieved an 87% accuracy rate. Further, in prior discussions they established that cropping images were not viable for their project due to the nature of the application.

- **Group 38**: Greetings and project introduction. The group went on to discuss how they spent the week comparing their programs training against a Deep Learn algorithm as a means of evaluation. The group went on to demonstrate their work and held a discussion in which they identified computational struggles. The group was advised to use the cluster and subsequently held a discussion on audio quality and means of enhancing their model's training effectiveness. This resulted in the advisement of grouping similar audio quality files together for training.

- **Group 06**: Greetings and project introduction. The group was experiencing issues with scheduling and identified that they were behind schedule. The group also identified issues with varying clap length which led to inaccuracies within the model. The group went on to discuss how they required more data for training and how they utilized a variational data encoder function as a generator to create more data. The group also mentioned an additional function related to networking for data generation. The group went on to demonstrate their progress and showed a live recording and playback of a clap and its associated echo response.

- **Group 29**: Greetings and project introduction. The group started off by recapping where they left off last week. The group went on to discuss how they worked on improving following distance and lane changing functionality. The group then demonstrated this through a video in which we can see that the lanes are highlighted using different colors and how the following distance was calculated. The group then went on to show the GUI that they had created and how they used scripting for the lane calculations. The demonstrated GUI was created using PYQT5 and received a round of applause.

- **Group 34**: Greetings and project introduction. The group started off by recapping where they left off last week. Then the group went on to demonstrate Google Trends vs Tweeter vs Mastadon graphs and went on to discuss data set sizes and processing of said data.

- The meeting was concluded and general dismissal occurred.

5. WEEK 6

Location: 2nd floor Chamber of Mines, EIE Seminar Room.

Date: 13 October 2023.

Time: 09:00 am

Adjournment: The meeting concluded at 11:06 a.m. There will be no further meetings scheduled beyond this one.

Submitted and approved by: Vukosi Mathebula (Week 6 Secretary) and Siphamandla Malaza (Week 6 Chair).

Attendance:

- Groups: 23G52, 23G38, 23G06, 23G24, 23G34, and 23G29,

- Supervisors: Estelle Trengove and Martin Bekker.

Agenda Items:

1. Each group will demonstrate their progress for week 5.
2. Groups to present their plans for the Open Day showcase.
3. Each group will address potential Open Day presentation issues and propose contingency plans.
4. Each group can ask up to two questions during the presentation to save time. Additional questions will be addressed after all groups have presented.

Opening:

- Formal introduction and welcoming by Siphamandla Malaza at 9:05 a.m.

Discussion

- **Group 52** stated that their objective for week 5 was to develop the user interface. They demonstrated how their application functions and how users can use it to convert black and white images into coloured images . The group explained how users can get access to their application using the QR code. The main challenge they encountered was when they tried to integrate their model into the application. They are currently working on converting their model into a format that React Native can recognize and have attempted to use the TensorFlow converter. However, they faced some issues during the process. The group mentioned that they are planning to start working on their poster for Open Day.

- **Group 38** mentioned that their objective was to transition from running their work on the CPU to using the cluster to access the GPU. However , they encountered issues with the compatibility of the GPU with Mozilla. They attempted to seeking help from a professor, but the problem could not be resolved. Furthermore, they realized that using GPU did not yield any significant improvements in terms of the computation speed, and it would still take them days to train their model. As a result, reverted to using the CPU. The group also presented the first version of their poster for open day.

- **Group 06** started by presenting their poster that they will use for the Open Day . They emphasized that their objective during the week was to make the averaging method work . To achieve this, they attempted trouble shooting techniques, which resulted into high unwanted frequencies, contrary to their expectations. Furthermore, they tried to add magnitude in the frequency domain when averaging and neglecting the phase, but this approach also failed. As an alternative plan, they intend to replace the averaging method with convolving claps with impulse response, and they expect this approach to work.

- **Group 24** presented their poster, which they were still working on for the Open Day. They also displayed the histogram graph they had created over the week, illustrating the distribution of happiness among people based on their financial statuses. They demonstrated how the histogram conveys information and explained how their model currently operates. The group emphasized that the level of wealth and happiness does not correlate. They also mentioned their plan on provide a high-level overview of their investigation to the audience during Open Day, and they explained how they will delve into the technical details if the audience is interested in learning more about their project. They also mentioned they intention to add more parameters to improve the performance of their model.

- **Group 34** mentioned that their aim during the week was to use the data they acquired about the events to establish patterns and trends from it. The group demonstrated their discoveries through graphs on Kaggle and provided a brief explanation of the results. The group emphasized the issue of having peaks at points where there were no significant events and how they were able to resolve it during the week . The group also stated that they plan to create their poster for Open Day before the beginning of the following week .

- **Group 29** stated that **t**heir primary objective of the week was to generate traffic data from different locations (footages), preprocess the data, and compare the driving cultures in these areas. They presented their results using tables and graphs that showed their initial findings and provided a brief explanation of their results. The group also explained their plan for testing their model's accuracy and documenting their results. In addition, they demonstrated both their initial poster and their improved poster, which they created after consulting with their supervisor. The group emphasized that certain features, including vehicle indicator detection, were not thoroughly implemented and sought advice on whether to include them in the report. They also asked questions about ethical considerations, and the social, economic and environmental impact of the investigation in the report. The supervisor suggested that they should not be included. Furthermore, they asked about the report format and whether they should demonstrate the experimental process they used to obtain the result.

- Supervisors clarified that they won't review reports anymore; they will only assess the structure of the report. They recommended students review each other's reports.

APPENDIX F: PRIVATE MEETINGS

1. WEEK 1

Location: MS Teams.
Date: 4 September 2023.
Time: 15:00 p.m.
Duration: 24 minutes
Adjournment: The meeting concluded at 15:24 p.m.
Attendance: Group 23G31 and supervisor Prof Vered Aharonson

Agenda Items:

1. Literature review discussion
2. Scope
3. Design discussion

Discussion:

During the meeting, our primary focus was centered on the presentation and discussion of our research. We collectively settled on a selection of key technologies, including SUMO for traffic simulation, reinforcement learning in a PettingZoo environment, and communication facilitation using Sumo-rl, Supersuit, and Traci. Our model, employing Proximal Policy Optimization (PPO), independently optimized traffic light timings for each intersection in a decentralized approach. Real-world maps were imported from OpenStreetMap, and traffic was generated using randomTrips.py. Professor Aharonson advised summarizing the literature review into concise paragraphs, clarified the non-compulsory nature of cohort meetings and the non-grading of notebooks, and suggested reaching out to our HOD for confirmation. Our meeting concluded with establishing a weekly meeting schedule for each Monday at 10:00 a.m.

2. WEEK 2

Location: MS Teams.
Date: 9 September 2023.
Time: 10:00 a.m.
Duration: 44 minutes
Adjournment: The meeting concluded at 10:44 a.m.
Attendance: Group 23G31 and supervisor Prof Vered Aharonson

Agenda Items:

1. Discuss the quality and feasibility of obtaining real world traffic data
2. Observation space
3. Optimization techniques

Discussion:

We presented our initial training results based on a tutorial found in the sumo-rl repository by Lucas Alegre. Prof Aharonson's guidance emphasized adherence to specific principles, including sustainability, simplicity, accuracy, scalability, and reliability. Our approach aimed to avoid unnecessary reinvention and instead, concentrate on enhancing pre-existing solutions or exploring uncharted territories. During our discussions, we explored strategies for optimizing the training process, which involved utilizing parallel environments, multiple CPU cores, and PettingZoo's SuperSuit wrapper. Additionally, we discussed the challenges of gaining access to the Waze for Cities program for real world traffic data. As an alternative, we examined the possibility of generating our own traffic data by employing the randomTrips.py script developed by the SUMO ellipse foundation.

3. WEEK 3

Location: MS Teams.
Date: 18 September 2023.
Time: 10:00 a.m.
Duration: 31 minutes
Adjournment: The meeting concluded at 10:31 p.m.
Attendance: Group 23G31 and supervisor Prof Vered Aharonson

Agenda Items:

1. Results and metrics
2. Benchmarks
3. Hyper parameter tuning

Discussion:

The meeting commenced with a review of the achieved success criteria. We had successfully trained a Multi-Agent Reinforcement Learning (MARL) model, utilizing both DQN and PPO algorithms on real-world maps. Training encompassed multiple simulations to enhance model performance and to further improve the learning process, custom observations and rewards had been implemented. In addition various benchmarks were generated for comparison, including a greedy solution and a camera monitoring system. Comprehensive result plots had been generated for in-depth analysis and deliberations had revolved around the model's compatibility with Waze or Google data through the exploration of converting Google intersection observations into SUMO-compatible formats. The meeting concluded with suggestions regarding additional results and metrics, including waiting time, cars stopped, speed, etc., and the incorporation of a training progress graph.

4. WEEK 4

Location: MS Teams.
Date: 4 September 2023.
Time: 15:00
Duration: 24 minutes
Adjournment: The meeting concluded at 15:24.
Attendance: Group 23G31 and supervisor Prof Vered Aharonson

Agenda Items:

1. Limitations
2. Observations and rewards3.
3. Parameter justification

Discussion:

The meeting commenced with a discussion on potential limitations of the existing system, such as a 1-second update time on SUMO and an 8-second minimum phase time due to driver reaction time. We explored various observation spaces, considering fixed, camera, GPS, and fully observable environments. For instance, GPS could only detect 77% of the cars on the road, while camera systems had a limited observation radius of 25 meters around the intersection. Additionally, we deliberated over different reward options, including wait time, speed, and pressure. Employing a difference-based reward system allowed the model to be both penalized and rewarded appropriately. We conducted an extensive evaluation involving 12 different observations and 11 distinct rewards across two maps, with each combination assessed using 12 distinct metrics. Ultimately, Observation 3 coupled with a reward based on the difference in wait times and speed emerged as the most effective approach.

5. WEEK 5

Location: MS Teams.
Date: 2 October 2023.
Time: 10:00 a.m.
Duration: 37 minutes
Adjournment: The meeting concluded at 10:37.
Attendance: Group 23G31 and supervisor Prof Vered Aharonson

Agenda Items:

1. Finalize results
2. Address issues

Discussion:

The meeting focused on the finalization of our results, which encompassed a total of 12 evaluation metrics, covering both simulation and training progress. These metrics allowed us to rank each model comprehensively across the entire map, system, or agent-controlled intersection. During the meeting, we addressed several issues and their corresponding resolutions, such as failed training, collisions, intersections remaining static due to unfavorable long-term rewards, file corruption, and naming schemes.

6. WEEK 6

Location: MS Teams.
Date: 4 September 2023.
Time: 15:00
Duration: 24 minutes
Adjournment: The meeting concluded at 15:24.
Attendance: Group 23G31 and supervisor Prof Vered Aharonson

Agenda Items:

1. Open day poster

Discussion:

The meeting centered on the presentation and discussion of our open day poster, focusing on its content and presentation style. We opted to create a digital A1 poster with all images generated using DALL E 3 and incorporated various figures and diagrams to effectively convey our message. Additionally, we decided to include mini traffic light LEDs controlled by a microcontroller. In addition, we generated QR links to various demos and simulations, along with a PowerPoint explaining the entire procedure. Vered expressed her enthusiasm about our intention to compete and win in the poster competition. It was discussed that there would be no further meetings.

APPENDIX G: LINKS

1. Github
https://github.com/Traffic-Light-Optimization/traffic-light-optimization

2. Poster
https://firebasestorage.googleapis.com/v0/b/investigation-project-33a2b.appspot.com/o/poster.pdf?alt=media&token=2b418daa-1714-43fe-a219-51f7747de9d4&_gl=1*gfoj2p*_ga*MTQxNTMwMzExMC4xNjkxNTA0MDM3*_ga_CW55HF8NVT*MTY5ODE2NTU1Ni4xOC4xLjE2OTgxNjU1OTguMTguMC4w

3. Powerpoint presentation
https://firebasestorage.googleapis.com/v0/b/investigation-project-33a2b.appspot.com/o/Traffic_Scenarios.pptx?alt=media&token=579105ba-c58e-4807-a4f8-2a93e470bb4d&_gl=1*1g8meiz*_ga*MTQxNTMwMzExMC4xNjkxNTA0MDM3*_ga_CW55HF8NVT*MTY5ODE2NTU1Ni4xOC4xLjE2OTgxNjY1NDYuNjAuMC4w

4. Demo video
https://firebasestorage.googleapis.com/v0/b/investigation-project-33a2b.appspot.com/o/demoMap.mp4?alt=media&token=f44df61d-0926-4048-8321-6630b626b2f3&_gl=1*1wvbk2o*_ga*MTQxNTMwMzExMC4xNjkxNTA0MDM3*_ga_CW55HF8NVT*MTY5ODE2NTU1Ni4xOC4xLjE2OTgxNjU2NTAuNjAuMC4w

https://firebasestorage.googleapis.com/v0/b/investigation-project-33a2b.appspot.com/o/comparison.mp4?alt=media&token=1f4cac2b-8f5b-4acf-b000-44b1296d4a0c&_gl=1*6ktizd*_ga*MTQxNTMwMzExMC4xNjkxNTA0MDM3*_ga_CW55HF8NVT*MTY5ODE2NTU1Ni4xOC4xLjE2OTgxNjU2NzUuMzUuMC4w

https://firebasestorage.googleapis.com/v0/b/investigation-project-33a2b.appspot.com/o/Beyers.mp4?alt=media&token=29c2d6f2-6c49-4f2e-bed4-68c924134188&_gl=1*96zr7u*_ga*MTQxNTMwMzExMC4xNjkxNTA0MDM3*_ga_CW55HF8NVT*MTY5ODE2NTU1Ni4xOC4xLjE2OTgxNjU2OTMuMTcuMC4w

APPENDIX H: CODE

1. Environment simulation

```
"""SUMO Environment for Traffic Signal Control."""
import os
import sys
from pathlib import Path
from typing import Callable, Optional, Tuple, Union
import random


if "SUMO_HOME" in os.environ:
    tools = os.path.join(os.environ["SUMO_HOME"], "tools")
    sys.path.append(tools)
else:
    raise ImportError("Please declare the environment variable 'SUMO_HOME'")
import gymnasium as gym
import numpy as np
import pandas as pd
import sumolib
import traci
from gymnasium.utils import EzPickle, seeding
from pettingzoo import AECEnv
from pettingzoo.utils import agent_selector, wrappers
from pettingzoo.utils.conversions import parallel_wrapper_fn

from .observations import DefaultObservationFunction, ObservationFunction
from .traffic_signal import TrafficSignal


LIBSUMO = "LIBSUMO_AS_TRACI" in os.environ


def env(**kwargs):
    """Instantiate a PettingoZoo environment."""
    env = SumoEnvironmentPZ(**kwargs)
    env = wrappers.AssertOutOfBoundsWrapper(env)
    env = wrappers.OrderEnforcingWrapper(env)
    return env


parallel_env = parallel_wrapper_fn(env)

    metadata = {
        "render_modes": ["human", "rgb_array"],
    }

    CONNECTION_LABEL = 0  # For traci multi-client support

    def __init__(
        self,
        net_file: str,
        route_file: str,
        out_csv_name: Optional[str] = None,
        use_gui: bool = False,
        virtual_display: Tuple[int, int] = (3200, 1800),
        begin_time: int = 0,
        num_seconds: int = 20000,
        max_depart_delay: int = -1,
        waiting_time_memory: int = 1000,
        time_to_teleport: int = -1,
        delta_time: int = 5,
        yellow_time: int = 2,
        min_green: int = 5,
        max_green: int = 50,
        single_agent: bool = False,
        reward_fn: Union[str, Callable, dict] = "diff-waiting-time",
        observation_class: ObservationFunction = DefaultObservationFunction,
        add_system_info: bool = True,
        add_per_agent_info: bool = True,
        sumo_seed: Union[str, int] = "random",
        fixed_ts: bool = False,
        sumo_warnings: bool = True,
        additional_sumo_cmd: Optional[str] = None,
        render_mode: Optional[str] = None,
        hide_cars: bool = False

    ) -> None:
        """Initialize the environment."""
        assert render_mode is None or render_mode in self.metadata["render_modes"], "Invalid render mode."
        self.render_mode = render_mode
        self.virtual_display = virtual_display
        self.disp = None

        self.hide_cars = hide_cars
        self._net = net_file
        self._route = route_file
        self.use_gui = use_gui
        if self.use_gui or self.render_mode is not None:
            self._sumo_binary = sumolib.checkBinary("sumo-gui")
        else:
            self._sumo_binary = sumolib.checkBinary("sumo")

        assert delta_time > yellow_time, "Time between actions must be at least greater than yellow time."

        self.begin_time = begin_time
        self.sim_max_time = begin_time + num_seconds
        self.delta_time = delta_time  # seconds on sumo at each step
        self.max_depart_delay = max_depart_delay  # Max wait time to insert a vehicle
        self.waiting_time_memory = waiting_time_memory  # Number of seconds to remember the waiting time of a vehicle (see https://sumo.dlr.de/pydoc/traci._vehicle.html#VehicleDomain-getAccumulatedWaitingTime)
        self.time_to_teleport = time_to_teleport
        self.min_green = min_green
        self.max_green = max_green
        self.yellow_time = yellow_time
        self.previous_accumulated_waiting_time = 0.0
        self.previous_agents_accumulated_waiting_time = 0.0
        self.single_agent = single_agent
        self.reward_fn = reward_fn
        self.sumo_seed = sumo_seed
        self.fixed_ts = fixed_ts
        self.sumo_warnings = sumo_warnings
        self.additional_sumo_cmd = additional_sumo_cmd
        self.add_system_info = add_system_info
        self.add_per_agent_info = add_per_agent_info
        self.label = str(SumoEnvironment.CONNECTION_LABEL)
        SumoEnvironment.CONNECTION_LABEL += 1
        self.sumo = None

        if LIBSUMO:
            traci.start([sumolib.checkBinary("sumo"), "-n", self._net])  # Start only to retrieve traffic light information
            conn = traci
        else:
            traci.start([sumolib.checkBinary("sumo"), "-n", self._net], label="init_connection" + self.label)
            conn = traci.getConnection("init_connection" + self.label)

        self.ts_ids = list(conn.trafficlight.getIDList())

        self.observation_class = observation_class

        if isinstance(self.reward_fn, dict):
            self.traffic_signals = {
                ts: TrafficSignal(
                    self,
                    ts,
                    self.delta_time,
                    self.yellow_time,
                    self.min_green,
                    self.max_green,
                    self.begin_time,
                    self.reward_fn[ts],
                    conn,
                )
                for ts in self.reward_fn.keys()
            }
        else:
            self.traffic_signals = {
                ts: TrafficSignal(
                    self,
                    ts,
```

```
                self.delta_time,
                self.yellow_time,
                self.min_green,
                self.max_green,
                self.begin_time,
                self.reward_fn,
                conn,
            )
            for ts in self.ts_ids
        }

        conn.close()

        self.vehicles = dict()
        self.reward_range = (-float("inf"), float("inf"))
        self.episode = 0
        self.metrics = []
        self.out_csv_name = out_csv_name
        self.observations = {ts: None for ts in self.ts_ids}
        self.rewards = {ts: None for ts in self.ts_ids}

    def _start_simulation(self):
        sumo_cmd = [
            self._sumo_binary,
            "-n",
            self._net,
            "-r",
            self._route,
            "--max-depart-delay",
            str(self.max_depart_delay),
            "--waiting-time-memory",
            str(self.waiting_time_memory),
            "--time-to-teleport",
            str(self.time_to_teleport),
        ]
        if self.begin_time > 0:
            sumo_cmd.append(f"-b {self.begin_time}")
        if self.sumo_seed == "random":
            sumo_cmd.append("--random")
        else:
            sumo_cmd.extend(["--seed", str(self.sumo_seed)])
        if not self.sumo_warnings:
            sumo_cmd.append("--no-warnings")
        if self.additional_sumo_cmd is not None:
            sumo_cmd.extend(self.additional_sumo_cmd.split())
        if self.use_gui or self.render_mode is not None:
            sumo_cmd.extend(["--start", "--quit-on-end"])
            if self.render_mode == "rgb_array":
                sumo_cmd.extend(["--window-size",
f"{self.virtual_display[0]},{self.virtual_display[1]}"])
                from pyvirtualdisplay.smartdisplay import SmartDisplay

                print("Creating a virtual display.")
                self.disp = SmartDisplay(size=self.virtual_display)
                self.disp.start()
                print("Virtual display started.")

        if LIBSUMO:
            traci.start(sumo_cmd)
            self.sumo = traci
        else:
            traci.start(sumo_cmd, label=self.label)
            self.sumo = traci.getConnection(self.label)

        if self.use_gui or self.render_mode is not None:
            self.sumo.gui.setSchema(traci.gui.DEFAULT_VIEW, "real world")


    ###TESTING
    def change_car_colours(self, probability):
        """Change the newly loaded cars colour based on the probability given"""
        new_cars = self.sumo.simulation.getDepartedIDList()
        for car in new_cars:
            if random.random() < probability:
                self.sumo.vehicle.setColor(car, (255, 255, 255, 255)) #white

    def reset(self, seed: Optional[int] = None, **kwargs):
        """Reset the environment."""
        super().reset(seed=seed, **kwargs)

        if self.episode != 0:
            self.close()
            self.save_csv(self.out_csv_name, self.episode)
        self.episode += 1
        self.metrics = []

        if seed is not None:
            self.sumo_seed = seed
        self._start_simulation()

        if isinstance(self.reward_fn, dict):
            self.traffic_signals = {
                ts: TrafficSignal(
                    self,
                    ts,
                    self.delta_time,
                    self.yellow_time,
                    self.min_green,
                    self.max_green,
                    self.begin_time,
                    self.reward_fn[ts],
                    self.sumo,
                )
                for ts in self.reward_fn.keys()
            }
        else:
            self.traffic_signals = {
                ts: TrafficSignal(
                    self,
                    ts,
                    self.delta_time,
                    self.yellow_time,
                    self.min_green,
                    self.max_green,
                    self.begin_time,
                    self.reward_fn,
                    self.sumo,
                )
                for ts in self.ts_ids
            }

        self.vehicles = dict()


        if self.single_agent:
            return self._compute_observations()[self.ts_ids[0]], self._compute_info()
        else:
            return self._compute_observations()

    @property
    def sim_step(self) -> float:
        """Return current simulation second on SUMO."""
        return self.sumo.simulation.getTime()

    def step(self, action: Union[dict, int]):
        """Apply the action(s) and then step the simulation for delta_time seconds.

        Args:
            action (Union[dict, int]): action(s) to be applied to the environment.
            If single_agent is True, action is an int, otherwise it expects a dict with keys corresponding
to traffic signal ids.
        """
        # No action, follow fixed TL defined in self.phases
        if action is None or action == {}:
            for _ in range(self.delta_time):
                self._sumo_step()
        else:
            self._apply_actions(action)
            self._run_steps()

        observations = self._compute_observations()
        rewards = self._compute_rewards()
        dones = self._compute_dones()
        terminated = False  # there are no 'terminal' states in this environment
        truncated = dones["__all__"]  # episode ends when sim_step >= max_steps
        info = self._compute_info()

        if self.single_agent:
            return observations[self.ts_ids[0]], rewards[self.ts_ids[0]], terminated, truncated, info
        else:
            return observations, rewards, dones, info
```

```
    def _run_steps(self):
        time_to_act = False
        while not time_to_act:
            self._sumo_step()
            for ts in self.ts_ids:
                self.traffic_signals[ts].update()
                if self.traffic_signals[ts].time_to_act:
                    time_to_act = True

    def _apply_actions(self, actions):
        """Set the next green phase for the traffic signals.

        Args:
            actions: If single-agent, actions is an int between 0 and self.num_green_phases (next green phase)
                     If multiagent, actions is a dict {ts_id : greenPhase}
        """
        if self.single_agent:
            if self.traffic_signals[self.ts_ids[0]].time_to_act:
                self.traffic_signals[self.ts_ids[0]].set_next_phase(actions)
        else:
            for ts, action in actions.items():
                if self.traffic_signals[ts].time_to_act:
                    self.traffic_signals[ts].set_next_phase(action)

    def _compute_dones(self):
        dones = {ts_id: False for ts_id in self.ts_ids}
        dones["__all__"] = self.sim_step >= self.sim_max_time
        return dones

    def _compute_info(self):
        info = {"step": self.sim_step}
        if self.add_system_info:
            info.update(self._get_system_info())
        if self.add_per_agent_info:
            info.update(self._get_per_agent_info())
        self.metrics.append(info.copy())
        return info

    def _compute_observations(self):
        self.observations.update(
            {ts: self.traffic_signals[ts].compute_observation() for ts in self.ts_ids if self.traffic_signals[ts].time_to_act}
        )
        return {ts: self.observations[ts].copy() for ts in self.observations.keys() if self.traffic_signals[ts].time_to_act}

    def _compute_rewards(self):
        self.rewards.update(
            {ts: self.traffic_signals[ts].compute_reward() for ts in self.ts_ids if self.traffic_signals[ts].time_to_act}
        )
        return {ts: self.rewards[ts] for ts in self.rewards.keys() if self.traffic_signals[ts].time_to_act}

    @property
    def observation_space(self):
        """Return the observation space of a traffic signal.

        Only used in case of single-agent environment.
        """
        return self.traffic_signals[self.ts_ids[0]].observation_space

    @property
    def action_space(self):
        """Return the action space of a traffic signal.

        Only used in case of single-agent environment.
        """
        return self.traffic_signals[self.ts_ids[0]].action_space

    def observation_spaces(self, ts_id: str):
        """Return the observation space of a traffic signal."""
        return self.traffic_signals[ts_id].observation_space

    def action_spaces(self, ts_id: str) -> gym.spaces.Discrete:
        """Return the action space of a traffic signal."""
        return self.traffic_signals[ts_id].action_space

    def num_agents(self):
        """Return number of agent."""
        return self.ts_ids
```

```
    def _sumo_step(self):
        if self.hide_cars:
            self.change_car_colours(0.23)
        colliding_vehicles = self.sumo.simulation.getCollidingVehiclesIDList()
        for vehicle in colliding_vehicles:
            try:
                if vehicle in self.sumo.vehicle.getIDList():
                    self.sumo.vehicle.remove(vehicle)
                    print(f"Removing vehicle {vehicle} due to collision")
            except Exception as e:
                print(f"Failed to remove collided vehicle {vehicle} due to exception {e}")
        self.sumo.simulationStep()

    def _get_system_info(self):
        vehicles = self.sumo.vehicle.getIDList()
        speeds = [self.sumo.vehicle.getSpeed(vehicle) for vehicle in vehicles]
        waiting_times = [self.sumo.vehicle.getWaitingTime(vehicle) for vehicle in vehicles]
        accumulated_waiting_times = [self.sumo.vehicle.getAccumulatedWaitingTime(vehicle) for vehicle in vehicles]
        system_accumulated_waiting_times = sum(accumulated_waiting_times)
        delta_system_accumulated_waiting_times = 0.0 if (system_accumulated_waiting_times - self.previous_accumulated_waiting_time) < 0 else system_accumulated_waiting_times - self.previous_accumulated_waiting_time
        self.previous_accumulated_waiting_time = system_accumulated_waiting_times
        return {
            # In SUMO, a vehicle is considered halting if its speed is below 0.1 m/s
            "system_total_stopped": sum(int(speed < 0.1) for speed in speeds),
            "system_total_waiting_time": sum(waiting_times),
            "system_mean_waiting_time": 0.0 if len(vehicles) == 0 else np.mean(waiting_times),
            "system_mean_speed": 0.0 if len(vehicles) == 0 else np.mean(speeds),
            "system_cars_present": len(vehicles),
            "system_accumulated_waiting_time (100)": system_accumulated_waiting_times,
            "system_accumulated_mean_waiting_time (100)": 0.0 if len(vehicles) == 0 else np.mean(accumulated_waiting_times),
            "system_accumulated_waiting_time (delta)": delta_system_accumulated_waiting_times,
            "system_accumulated_mean_waiting_time (delta)": 0.0 if len(vehicles) == 0 else delta_system_accumulated_waiting_times/len(vehicles),
        }

    def _get_per_agent_info(self):
        stopped = [self.traffic_signals[ts].get_total_queued() for ts in self.ts_ids]
        accumulated_waiting_time = [
            sum(self.traffic_signals[ts].get_accumulated_waiting_time_per_lane()) for ts in self.ts_ids
        ]
        average_speed = [self.traffic_signals[ts].get_average_speed() for ts in self.ts_ids]
        agents_accumulated_waiting_time = sum(accumulated_waiting_time)
        delta_agents_accumulated_waiting_time = 0.0 if (agents_accumulated_waiting_time - self.previous_agents_accumulated_waiting_time) < 0 else agents_accumulated_waiting_time - self.previous_agents_accumulated_waiting_time
        self.previous_agents_accumulated_waiting_time =  agents_accumulated_waiting_time

        info = {}
        info["agents_total_stopped"] = sum(stopped)
        info["agents_total_accumulated_waiting_time (100)"] = agents_accumulated_waiting_time
        info["agents_total_accumulated_waiting_time (delta)"] = delta_agents_accumulated_waiting_time
        speed_list = []
        wait_list = []
        for ts in self.ts_ids:
            speed_list = speed_list + self.traffic_signals[ts].get_average_speed_list()
            wait_list = wait_list  + self.traffic_signals[ts].get_accumulated_waiting_time_list()
        info["agents_mean_speed"] = np.mean(speed_list)
        info["agents_mean_waiting_time (100)"] = np.mean(wait_list)
        #agents_mean_waiting_time (delta) does not work
        info["agents_mean_waiting_time (delta)"] = np.mean(wait_list)
        info["agents_cars_present"] = sum([sum(self.traffic_signals[ts].get_cars_present()) for ts in self.ts_ids])
        for i, ts in enumerate(self.ts_ids):
            info[f"{ts}_stopped"] = stopped[i]
            info[f"{ts}_accumulated_waiting_time"] = accumulated_waiting_time[i]
            info[f"{ts}_average_speed"] = average_speed[i]

        return info

    def close(self):
        """Close the environment and stop the SUMO simulation."""
        if self.sumo is None:
            return

        if not LIBSUMO:
```

```
            traci.switch(self.label)
        traci.close()

        if self.disp is not None:
            self.disp.stop()
            self.disp = None

        self.sumo = None

    def __del__(self):
        """Close the environment and stop the SUMO simulation."""
        self.close()

    def render(self):
        """Render the environment.

        If render_mode is "human", the environment will be rendered in a GUI window using pyvirtualdisplay.
        """
        if self.render_mode == "human":
            return  # sumo-gui will already be rendering the frame
        elif self.render_mode == "rgb_array":
            # img = self.sumo.gui.screenshot(traci.gui.DEFAULT_VIEW,
            #                                f"temp/img{self.sim_step}.jpg",
            #                                width=self.virtual_display[0],
            #                                height=self.virtual_display[1])
            img = self.disp.grab()
            return np.array(img)

    def save_csv(self, out_csv_name, episode):
        """Save metrics of the simulation to a .csv file.

        Args:
            out_csv_name (str): Path to the output .csv file. E.g.: "results/my_results
            episode (int): Episode number to be appended to the output file name.
        """
        if out_csv_name is not None:
            df = pd.DataFrame(self.metrics)
            Path(Path(out_csv_name).parent).mkdir(parents=True, exist_ok=True)
            df.to_csv(out_csv_name + f"_conn{self.label}_ep{episode}" + ".csv", index=False)

    # Below functions are for discrete state space

    def encode(self, state, ts_id):
        """Encode the state of the traffic signal into a hashable object."""
        phase = int(np.where(state[: self.traffic_signals[ts_id].num_green_phases] == 1)[0])
        min_green = state[self.traffic_signals[ts_id].num_green_phases]
        density_queue = [self._discretize_density(d) for d in state[self.traffic_signals[ts_id].num_green_phases + 1 :]]
        # tuples are hashable and can be used as key in python dictionary
        return tuple([phase, min_green] + density_queue)

    def _discretize_density(self, density):
        return min(int(density * 10), 9)


class SumoEnvironmentPZ(AECEnv, EzPickle):
    """A wrapper for the SUMO environment that implements the AECEnv interface from PettingZoo.

    For more information, see https://pettingzoo.farama.org/api/aec/.

    The arguments are the same as for :py:class:`sumo_rl.environment.env.SumoEnvironment`.
    """

    metadata = {"render.modes": ["human", "rgb_array"], "name": "sumo_rl_v0", "is_parallelizable": True}

    def __init__(self, **kwargs):
        """Initialize the environment."""
        EzPickle.__init__(self, **kwargs)
        self._kwargs = kwargs

        self.seed()
        self.env = SumoEnvironment(**self._kwargs)

        self.agents = self.env.ts_ids
        self.possible_agents = self.env.ts_ids
        self._agent_selector = agent_selector(self.agents)
        self.agent_selection = self._agent_selector.reset()
        # spaces
        self.action_spaces = {a: self.env.action_spaces(a) for a in self.agents}
        self.observation_spaces = {a: self.env.observation_spaces(a) for a in self.agents}

        # dicts
        self.rewards = {a: 0 for a in self.agents}
        self.terminations = {a: False for a in self.agents}
        self.truncations = {a: False for a in self.agents}
        self.infos = {a: {} for a in self.agents}

    def seed(self, seed=None):
        """Set the seed for the environment."""
        self.randomizer, seed = seeding.np_random(seed)

    def reset(self, seed: Optional[int] = None, options: Optional[dict] = None):
        """Reset the environment."""
        self.env.reset(seed=seed, options=options)
        self.agents = self.possible_agents[:]
        self.agent_selection = self._agent_selector.reset()
        self.rewards = {agent: 0 for agent in self.agents}
        self._cumulative_rewards = {agent: 0 for agent in self.agents}
        self.terminations = {a: False for a in self.agents}
        self.truncations = {a: False for a in self.agents}
        self.compute_info()

    def compute_info(self):
        """Compute the info for the current step."""
        self.infos = {a: {} for a in self.agents}
        infos = self.env._compute_info()
        for a in self.agents:
            for k, v in infos.items():
                if k.startswith(a) or k.startswith("system"):
                    self.infos[a][k] = v

    def observation_space(self, agent):
        """Return the observation space for the agent."""
        return self.observation_spaces[agent]

    def num_agents(self):
        """Return number of agent."""
        return self.agents

    def action_space(self, agent):
        """Return the action space for the agent."""
        return self.action_spaces[agent]

    def observe(self, agent):
        """Return the observation for the agent."""
        obs = self.env.observations[agent].copy()
        return obs

    def close(self):
        """Close the environment and stop the SUMO simulation."""
        self.env.close()

    def render(self):
        """Render the environment."""
        return self.env.render()

    def save_csv(self, out_csv_name, episode):
        """Save metrics of the simulation to a .csv file."""
        self.env.save_csv(out_csv_name, episode)

    def step(self, action):
        """Step the environment."""
        if self.truncations[self.agent_selection] or self.terminations[self.agent_selection]:
            return self._was_dead_step(action)
        agent = self.agent_selection
        if not self.action_spaces[agent].contains(action):
            raise Exception(
                "Action for agent {} must be in Discrete({})."
                "It is currently {}".format(agent, self.action_spaces[agent].n, action)
            )

        self.env._apply_actions({agent: action})

        if self._agent_selector.is_last():
            self.env._run_steps()
            self.env._compute_observations()
            self.rewards = self.env._compute_rewards()
            self.compute_info()
        else:
```

```
            self._clear_rewards()

        done = self.env._compute_dones()["__all__"]
        self.truncations = {a: done for a in self.agents}
```

```
        self.agent_selection = self._agent_selector.next()
        self._cumulative_rewards[agent] = 0
        self._accumulate_rewards()g
```

2. Traffic signal controller

```
"""SUMO Environment for Traffic Signal Control."""
import os
import sys
from pathlib import Path
from typing import Callable, Optional, Tuple, Union
import random


if "SUMO_HOME" in os.environ:
    tools = os.path.join(os.environ["SUMO_HOME"], "tools")
    sys.path.append(tools)
else:
    raise ImportError("Please declare the environment variable 'SUMO_HOME'")
import gymnasium as gym
import numpy as np
import pandas as pd
import sumolib
import traci
from gymnasium.utils import EzPickle, seeding
from pettingzoo import AECEnv
from pettingzoo.utils import agent_selector, wrappers
from pettingzoo.utils.conversions import parallel_wrapper_fn

from .observations import DefaultObservationFunction, ObservationFunction
from .traffic_signal import TrafficSignal


LIBSUMO = "LIBSUMO_AS_TRACI" in os.environ


def env(**kwargs):
    """Instantiate a PettingoZoo environment."""
    env = SumoEnvironmentPZ(**kwargs)
    env = wrappers.AssertOutOfBoundsWrapper(env)
    env = wrappers.OrderEnforcingWrapper(env)
    return env


parallel_env = parallel_wrapper_fn(env)


class SumoEnvironment(gym.Env):
    metadata = {
        "render_modes": ["human", "rgb_array"],
    }

    CONNECTION_LABEL = 0  # For traci multi-client support

    def __init__(
        self,
        net_file: str,
        route_file: str,
        out_csv_name: Optional[str] = None,
        use_gui: bool = False,
        virtual_display: Tuple[int, int] = (3200, 1800),
        begin_time: int = 0,
        num_seconds: int = 20000,
        max_depart_delay: int = -1,
        waiting_time_memory: int = 1000,
        time_to_teleport: int = -1,
        delta_time: int = 5,
        yellow_time: int = 2,
        min_green: int = 5,
        max_green: int = 50,
        single_agent: bool = False,
        reward_fn: Union[str, Callable, dict] = "diff-waiting-time",
        observation_class: ObservationFunction = DefaultObservationFunction,
        add_system_info: bool = True,
        add_per_agent_info: bool = True,
        sumo_seed: Union[str, int] = "random",
        fixed_ts: bool = False,
        sumo_warnings: bool = True,
        additional_sumo_cmd: Optional[str] = None,
        render_mode: Optional[str] = None,
        hide_cars: bool = False

    ) -> None:
        """Initialize the environment."""
        assert render_mode is None or render_mode in self.metadata["render_modes"], "Invalid render
mode."
        self.render_mode = render_mode
        self.virtual_display = virtual_display
        self.disp = None

        self.hide_cars = hide_cars
        self._net = net_file
        self._route = route_file
        self.use_gui = use_gui
        if self.use_gui or self.render_mode is not None:
            self._sumo_binary = sumolib.checkBinary("sumo-gui")
        else:
            self._sumo_binary = sumolib.checkBinary("sumo")

        assert delta_time > yellow_time, "Time between actions must be at least greater than yellow
time."

        self.begin_time = begin_time
        self.sim_max_time = begin_time + num_seconds
        self.delta_time = delta_time  # seconds on sumo at each step
        self.max_depart_delay = max_depart_delay  # Max wait time to insert a vehicle
        self.waiting_time_memory = waiting_time_memory  # Number of seconds to remember the
waiting time of a vehicle (see
https://sumo.dlr.de/pydoc/traci._vehicle.html#VehicleDomain-getAccumulatedWaitingTime)
        self.time_to_teleport = time_to_teleport
        self.min_green = min_green
        self.max_green = max_green
        self.yellow_time = yellow_time
        self.previous_accumulated_waiting_time = 0.0
        self.previous_agents_accumulated_waiting_time = 0.0
        self.single_agent = single_agent
        self.reward_fn = reward_fn
        self.sumo_seed = sumo_seed
        self.fixed_ts = fixed_ts
        self.sumo_warnings = sumo_warnings
        self.additional_sumo_cmd = additional_sumo_cmd
        self.add_system_info = add_system_info
        self.add_per_agent_info = add_per_agent_info
        self.label = str(SumoEnvironment.CONNECTION_LABEL)
        SumoEnvironment.CONNECTION_LABEL += 1
        self.sumo = None

        if LIBSUMO:
            traci.start([sumolib.checkBinary("sumo"), "-n", self._net])  # Start only to retrieve traffic
light information
            conn = traci
        else:
            traci.start([sumolib.checkBinary("sumo"), "-n", self._net], label="init_connection" +
self.label)
            conn = traci.getConnection("init_connection" + self.label)

        self.ts_ids = list(conn.trafficlight.getIDList())

        self.observation_class = observation_class

        if isinstance(self.reward_fn, dict):
            self.traffic_signals = {
                ts: TrafficSignal(
                    self,
```

```
                ts,
                self.delta_time,
                self.yellow_time,
                self.min_green,
                self.max_green,
                self.begin_time,
                self.reward_fn[ts],
                conn,
            )
            for ts in self.reward_fn.keys()
        }
    else:
        self.traffic_signals = {
            ts: TrafficSignal(
                self,
                ts,
                self.delta_time,
                self.yellow_time,
                self.min_green,
                self.max_green,
                self.begin_time,
                self.reward_fn,
                conn,
            )
            for ts in self.ts_ids
        }

    conn.close()

    self.vehicles = dict()
    self.reward_range = (-float("inf"), float("inf"))
    self.episode = 0
    self.metrics = []
    self.out_csv_name = out_csv_name
    self.observations = {ts: None for ts in self.ts_ids}
    self.rewards = {ts: None for ts in self.ts_ids}

def _start_simulation(self):
    sumo_cmd = [
        self._sumo_binary,
        "-n",
        self._net,
        "-r",
        self._route,
        "--max-depart-delay",
        str(self.max_depart_delay),
        "--waiting-time-memory",
        str(self.waiting_time_memory),
        "--time-to-teleport",
        str(self.time_to_teleport),
    ]
    if self.begin_time > 0:
        sumo_cmd.append(f"-b {self.begin_time}")
    if self.sumo_seed == "random":
        sumo_cmd.append("--random")
    else:
        sumo_cmd.extend(["--seed", str(self.sumo_seed)])
    if not self.sumo_warnings:
        sumo_cmd.append("--no-warnings")
    if self.additional_sumo_cmd is not None:
        sumo_cmd.extend(self.additional_sumo_cmd.split())
    if self.use_gui or self.render_mode is not None:
        sumo_cmd.extend(["--start", "--quit-on-end"])
        if self.render_mode == "rgb_array":
            sumo_cmd.extend(["--window-size",
f"{self.virtual_display[0]},{self.virtual_display[1]}"])
            from pyvirtualdisplay.smartdisplay import SmartDisplay

            print("Creating a virtual display.")
            self.disp = SmartDisplay(size=self.virtual_display)
            self.disp.start()
            print("Virtual display started.")

    if LIBSUMO:
        traci.start(sumo_cmd)
        self.sumo = traci
    else:
        traci.start(sumo_cmd, label=self.label)
        self.sumo = traci.getConnection(self.label)

    if self.use_gui or self.render_mode is not None:
        self.sumo.gui.setSchema(traci.gui.DEFAULT_VIEW, "real world")


###TESTING
def change_car_colours(self, probability):
    """Change the newly loaded cars colour based on the probability given"""
    new_cars = self.sumo.simulation.getDepartedIDList()
    for car in new_cars:
        if random.random() < probability:
            self.sumo.vehicle.setColor(car, (255, 255, 255, 255)) #white

def reset(self, seed: Optional[int] = None, **kwargs):
    """Reset the environment."""
    super().reset(seed=seed, **kwargs)

    if self.episode != 0:
        self.close()
        self.save_csv(self.out_csv_name, self.episode)
    self.episode += 1
    self.metrics = []

    if seed is not None:
        self.sumo_seed = seed
    self._start_simulation()

    if isinstance(self.reward_fn, dict):
        self.traffic_signals = {
            ts: TrafficSignal(
                self,
                ts,
                self.delta_time,
                self.yellow_time,
                self.min_green,
                self.max_green,
                self.begin_time,
                self.reward_fn[ts],
                self.sumo,
            )
            for ts in self.reward_fn.keys()
        }
    else:
        self.traffic_signals = {
            ts: TrafficSignal(
                self,
                ts,
                self.delta_time,
                self.yellow_time,
                self.min_green,
                self.max_green,
                self.begin_time,
                self.reward_fn,
                self.sumo,
            )
            for ts in self.ts_ids
        }

    self.vehicles = dict()


    if self.single_agent:
        return self._compute_observations()[self.ts_ids[0]], self._compute_info()
    else:
        return self._compute_observations()

@property
def sim_step(self) -> float:
    """Return current simulation second on SUMO."""
    return self.sumo.simulation.getTime()

def step(self, action: Union[dict, int]):
    """Apply the action(s) and then step the simulation for delta_time seconds.

    Args:
        action (Union[dict, int]): action(s) to be applied to the environment.
        If single_agent is True, action is an int, otherwise it expects a dict with keys corresponding
to traffic signal ids.
    """
    # No action, follow fixed TL defined in self.phases
    if action is None or action == {}:
        for _ in range(self.delta_time):
            self._sumo_step()
```

```
        else:
            self._apply_actions(action)
            self._run_steps()

        observations = self._compute_observations()
        rewards = self._compute_rewards()
        dones = self._compute_dones()
        terminated = False  # there are no 'terminal' states in this environment
        truncated = dones["__all__"]  # episode ends when sim_step >= max_steps
        info = self._compute_info()

        if self.single_agent:
            return observations[self.ts_ids[0]], rewards[self.ts_ids[0]], terminated, truncated, info
        else:
            return observations, rewards, dones, info

    def _run_steps(self):
        time_to_act = False
        while not time_to_act:
            self._sumo_step()
            for ts in self.ts_ids:
                self.traffic_signals[ts].update()
                if self.traffic_signals[ts].time_to_act:
                    time_to_act = True

    def _apply_actions(self, actions):
        """Set the next green phase for the traffic signals.

        Args:
            actions: If single-agent, actions is an int between 0 and self.num_green_phases (next green
phase)
                If multiagent, actions is a dict {ts_id : greenPhase}
        """
        if self.single_agent:
            if self.traffic_signals[self.ts_ids[0]].time_to_act:
                self.traffic_signals[self.ts_ids[0]].set_next_phase(actions)
        else:
            for ts, action in actions.items():
                if self.traffic_signals[ts].time_to_act:
                    self.traffic_signals[ts].set_next_phase(action)

    def _compute_dones(self):
        dones = {ts_id: False for ts_id in self.ts_ids}
        dones["__all__"] = self.sim_step >= self.sim_max_time
        return dones

    def _compute_info(self):
        info = {"step": self.sim_step}
        if self.add_system_info:
            info.update(self._get_system_info())
        if self.add_per_agent_info:
            info.update(self._get_per_agent_info())
        self.metrics.append(info.copy())
        return info

    def _compute_observations(self):
        self.observations.update(
            {ts: self.traffic_signals[ts].compute_observation() for ts in self.ts_ids if
self.traffic_signals[ts].time_to_act}
        )
        return {ts: self.observations[ts].copy() for ts in self.observations.keys() if
self.traffic_signals[ts].time_to_act}

    def _compute_rewards(self):
        self.rewards.update(
            {ts: self.traffic_signals[ts].compute_reward() for ts in self.ts_ids if
self.traffic_signals[ts].time_to_act}
        )
        return {ts: self.rewards[ts] for ts in self.rewards.keys() if self.traffic_signals[ts].time_to_act}

    @property
    def observation_space(self):
        """Return the observation space of a traffic signal.

        Only used in case of single-agent environment.
        """
        return self.traffic_signals[self.ts_ids[0]].observation_space

    @property
    def action_space(self):
        """Return the action space of a traffic signal.

        Only used in case of single-agent environment.
        """
        return self.traffic_signals[self.ts_ids[0]].action_space

    def observation_spaces(self, ts_id: str):
        """Return the observation space of a traffic signal."""
        return self.traffic_signals[ts_id].observation_space

    def action_spaces(self, ts_id: str) -> gym.spaces.Discrete:
        """Return the action space of a traffic signal."""
        return self.traffic_signals[ts_id].action_space

    def num_agents(self):
        """Return number of agent."""
        return self.ts_ids

    def _sumo_step(self):
        if self.hide_cars:
            self.change_car_colours(0.23)
        colliding_vehicles = self.sumo.simulation.getCollidingVehiclesIDList()
        for vehicle in colliding_vehicles:
            try:
                if vehicle in self.sumo.vehicle.getIDList():
                    self.sumo.vehicle.remove(vehicle)
                    print(f"Removing vehicle {vehicle} due to collision")
            except Exception as e:
                print(f"Failed to remove collided vehicle {vehicle} due to exception {e}")
        self.sumo.simulationStep()

    def _get_system_info(self):
        vehicles = self.sumo.vehicle.getIDList()
        speeds = [self.sumo.vehicle.getSpeed(vehicle) for vehicle in vehicles]
        waiting_times = [self.sumo.vehicle.getWaitingTime(vehicle) for vehicle in vehicles]
        accumulated_waiting_times = [self.sumo.vehicle.getAccumulatedWaitingTime(vehicle) for
vehicle in vehicles]
        system_accumulated_waiting_times = sum(accumulated_waiting_times)
        delta_system_accumulated_waiting_times = 0.0 if (system_accumulated_waiting_times -
self.previous_accumulated_waiting_time) < 0 else system_accumulated_waiting_times -
self.previous_accumulated_waiting_time
        self.previous_accumulated_waiting_time = system_accumulated_waiting_times
        return {
            # In SUMO, a vehicle is considered halting if its speed is below 0.1 m/s
            "system_total_stopped": sum(int(speed < 0.1) for speed in speeds),
            "system_total_waiting_time": sum(waiting_times),
            "system_mean_waiting_time": 0.0 if len(vehicles) == 0 else np.mean(waiting_times),
            "system_mean_speed": 0.0 if len(vehicles) == 0 else np.mean(speeds),
            "system_cars_present": len(vehicles),
            "system_accumulated_waiting_time (100)": system_accumulated_waiting_times,
            "system_accumulated_mean_waiting_time (100)": 0.0 if len(vehicles) == 0 else
np.mean(accumulated_waiting_times),
            "system_accumulated_waiting_time (delta)": delta_system_accumulated_waiting_times,
            "system_accumulated_mean_waiting_time (delta)": 0.0 if len(vehicles) == 0 else
delta_system_accumulated_waiting_times/len(vehicles),
        }

    def _get_per_agent_info(self):
        stopped = [self.traffic_signals[ts].get_total_queued() for ts in self.ts_ids]
        accumulated_waiting_time = [
            sum(self.traffic_signals[ts].get_accumulated_waiting_time_per_lane()) for ts in self.ts_ids
        ]
        average_speed = [self.traffic_signals[ts].get_average_speed() for ts in self.ts_ids]
        agents_accumulated_waiting_time = sum(accumulated_waiting_time)
        delta_agents_accumulated_waiting_time = 0.0 if (agents_accumulated_waiting_time -
self.previous_agents_accumulated_waiting_time) < 0 else agents_accumulated_waiting_time -
self.previous_agents_accumulated_waiting_time
        self.previous_agents_accumulated_waiting_time =  agents_accumulated_waiting_time

        info = {}
        info["agents_total_stopped"] = sum(stopped)
        info["agents_total_accumulated_waiting_time (100)"] = agents_accumulated_waiting_time
        info["agents_total_accumulated_waiting_time (delta)"] =
delta_agents_accumulated_waiting_time
        speed_list = []
        wait_list = []
        for ts in self.ts_ids:
            speed_list = speed_list + self.traffic_signals[ts].get_average_speed_list()
            wait_list = wait_list  + self.traffic_signals[ts].get_accumulated_waiting_time_list()
        info["agents_mean_speed"] = np.mean(speed_list)
        info["agents_mean_waiting_time (100)"] = np.mean(wait_list)
        #agents_mean_waiting_time (delta) does not work
```

```
        info["agents_mean_waiting_time (delta)"] = np.mean(wait_list)
        info["agents_cars_present"] = sum([sum(self.traffic_signals[ts].get_cars_present()) for ts in
self.ts_ids])
        for i, ts in enumerate(self.ts_ids):
            info[f"{ts}_stopped"] = stopped[i]
            info[f"{ts}_accumulated_waiting_time"] = accumulated_waiting_time[i]
            info[f"{ts}_average_speed"] = average_speed[i]

        return info

    def close(self):
        """Close the environment and stop the SUMO simulation."""
        if self.sumo is None:
            return

        if not LIBSUMO:
            traci.switch(self.label)
        traci.close()

        if self.disp is not None:
            self.disp.stop()
            self.disp = None

        self.sumo = None

    def __del__(self):
        """Close the environment and stop the SUMO simulation."""
        self.close()

    def render(self):
        """Render the environment.

        If render_mode is "human", the environment will be rendered in a GUI window using
pyvirtualdisplay.
        """
        if self.render_mode == "human":
            return  # sumo-gui will already be rendering the frame
        elif self.render_mode == "rgb_array":
            # img = self.sumo.gui.screenshot(traci.gui.DEFAULT_VIEW,
            #                                f"temp/img{self.sim_step}.jpg",
            #                                width=self.virtual_display[0],
            #                                height=self.virtual_display[1])
            img = self.disp.grab()
            return np.array(img)

    def save_csv(self, out_csv_name, episode):
        """Save metrics of the simulation to a .csv file.

        Args:
            out_csv_name (str): Path to the output .csv file. E.g.: "results/my_results
            episode (int): Episode number to be appended to the output file name.
        """
        if out_csv_name is not None:
            df = pd.DataFrame(self.metrics)
            Path(Path(out_csv_name).parent).mkdir(parents=True, exist_ok=True)
            df.to_csv(out_csv_name + f"_conn{self.label}_ep{episode}" + ".csv", index=False)

    # Below functions are for discrete state space

    def encode(self, state, ts_id):
        """Encode the state of the traffic signal into a hashable object."""
        phase = int(np.where(state[: self.traffic_signals[ts_id].num_green_phases] == 1)[0])
        min_green = state[self.traffic_signals[ts_id].num_green_phases]
        density_queue = [self._discretize_density(d) for d in
state[self.traffic_signals[ts_id].num_green_phases + 1 :]]
        # tuples are hashable and can be used as key in python dictionary
        return tuple([phase, min_green] + density_queue)

    def _discretize_density(self, density):
        return min(int(density * 10), 9)


class SumoEnvironmentPZ(AECEnv, EzPickle):
    """A wrapper for the SUMO environment that implements the AECEnv interface from
PettingZoo.

    For more information, see https://pettingzoo.farama.org/api/aec/.

    The arguments are the same as for :py:class:`sumo_rl.environment.env.SumoEnvironment`.
    """

    metadata = {"render.modes": ["human", "rgb_array"], "name": "sumo_rl_v0", "is_parallelizable":
True}

    def __init__(self, **kwargs):
        """Initialize the environment."""
        EzPickle.__init__(self, **kwargs)
        self._kwargs = kwargs

        self.seed()
        self.env = SumoEnvironment(**self._kwargs)

        self.agents = self.env.ts_ids
        self.possible_agents = self.env.ts_ids
        self._agent_selector = agent_selector(self.agents)
        self.agent_selection = self._agent_selector.reset()
        # spaces
        self.action_spaces = {a: self.env.action_spaces(a) for a in self.agents}
        self.observation_spaces = {a: self.env.observation_spaces(a) for a in self.agents}

        # dicts
        self.rewards = {a: 0 for a in self.agents}
        self.terminations = {a: False for a in self.agents}
        self.truncations = {a: False for a in self.agents}
        self.infos = {a: {} for a in self.agents}

    def seed(self, seed=None):
        """Set the seed for the environment."""
        self.randomizer, seed = seeding.np_random(seed)

    def reset(self, seed: Optional[int] = None, options: Optional[dict] = None):
        """Reset the environment."""
        self.env.reset(seed=seed, options=options)
        self.agents = self.possible_agents[:]
        self.agent_selection = self._agent_selector.reset()
        self.rewards = {agent: 0 for agent in self.agents}
        self._cumulative_rewards = {agent: 0 for agent in self.agents}
        self.terminations = {a: False for a in self.agents}
        self.truncations = {a: False for a in self.agents}
        self.compute_info()

    def compute_info(self):
        """Compute the info for the current step."""
        self.infos = {a: {} for a in self.agents}
        infos = self.env._compute_info()
        for a in self.agents:
            for k, v in infos.items():
                if k.startswith(a) or k.startswith("system"):
                    self.infos[a][k] = v

    def observation_space(self, agent):
        """Return the observation space for the agent."""
        return self.observation_spaces[agent]

    def num_agents(self):
        """Return number of agent."""
        return self.agents

    def action_space(self, agent):
        """Return the action space for the agent."""
        return self.action_spaces[agent]

    def observe(self, agent):
        """Return the observation for the agent."""
        obs = self.env.observations[agent].copy()
        return obs

    def close(self):
        """Close the environment and stop the SUMO simulation."""
        self.env.close()

    def render(self):
        """Render the environment."""
        return self.env.render()

    def save_csv(self, out_csv_name, episode):
        """Save metrics of the simulation to a .csv file."""
        self.env.save_csv(out_csv_name, episode)

    def step(self, action):
        """Step the environment."""
        if self.truncations[self.agent_selection] or self.terminations[self.agent_selection]:
```

```
            return self._was_dead_step(action)
        agent = self.agent_selection
        if not self.action_spaces[agent].contains(action):
            raise Exception(
                "Action for agent {} must be in Discrete({})."
                "It is currently {}".format(agent, self.action_spaces[agent].n, action)
            )

        self.env._apply_actions({agent: action})

        if self._agent_selector.is_last():
            self.env._run_steps()
            self.env._compute_observations()
            self.rewards = self.env._compute_rewards()
            self.compute_info()
        else:
            self._clear_rewards()

        done = self.env._compute_dones()["__all__"]
        self.truncations = {a: done for a in self.agents}

        self.agent_selection = self._agent_selector.next()
        self._cumulative_rewards[agent] = 0
        self._accumulate_rewards()
```

3. Marl training script

```
from stable_baselines3 import PPO
from stable_baselines3 import DQN
from stable_baselines3.common.vec_env import VecMonitor
from stable_baselines3.common.callbacks import EvalCallback
import supersuit as ss
import sumo_rl
from supersuit.multiagent_wrappers import pad_observations_v0
from supersuit.multiagent_wrappers import pad_action_space_v0
from config_files.observation_class_directories import get_observation_class
from config_files.net_route_directories import get_file_locations
from config_files.delete_results import deleteTrainingResults
from config_files import reward_directories

# PARAMETERS
#=======================
# In each timestep (delta_time), the agent takes an action, and the environment (the traffic simulation) advances by delta_time seconds.
# The agent continues to take actions for total_timesteps.
# The simulation repeats and improves on the previous model by repeating the simulation for a number of episodes

numSeconds = 3600 # This parameter determines the total duration of the SUMO traffic simulation in seconds.
deltaTime = 8 #This parameter determines how much time in the simulation passes with each step.
max_green = 60
simRepeats = 40 # Number of episodes
parallelEnv = 12
# evaluation_interval = 500 #How many seconds in you want to evaluate the model that is being trained to save the best one
num_cpus = 4
yellow_time = 3 # min yellow time
totalTimesteps = numSeconds*simRepeats*parallelEnv # This is the total number of steps in the environment that the agent will take for training. It's the overall budget of steps that the agent can interact with the environment.
maps = ["cologne1", "cologne3", "cologne8"]
mdl = 'PPO' # Set to DQN for DQN model
observations = ["ideal", "camera", "gps"]
seed = '12345' # or 'random' '8493'  1234 99393
gui = False # Set to True to see the SUMO-GUI

# START TRAINING
# =======================
if __name__ == "__main__":

    for map in maps:
      net_route_files = get_file_locations(map) # Select a map

      for observation in observations:
          #Reward
          reward_option = 'defandspeed' if observation != 'gps' else 'defandspeedwithmaxgreen'

          #Model save path
          model_save_path = f"./models/{map}_{mdl}_{observation}_{reward_option}"

          #Delete results
          deleteTrainingResults(map, mdl, observation, reward_option)

          #Get observation class
          observation_class =  get_observation_class("model", observation)

          # Get the corresponding reward function based on the option
          reward_function = reward_directories.reward_functions.get(reward_option)


          results_path =
f'./results/marl_train/marl_train-{map}-{mdl}-{observation}-{reward_option}'
          print(results_path)

          # creates a SUMO environment with multiple intersections, each controlled by a separate agent.
          env = sumo_rl.parallel_env(
              net_file=net_route_files["net"],
              route_file=net_route_files["route"],
              use_gui=gui,
              num_seconds=numSeconds,
              delta_time=deltaTime,
              max_green=max_green,
              out_csv_name=results_path,
              sumo_seed = seed,
              yellow_time = yellow_time,
              reward_fn=reward_function,
              add_per_agent_info = True,
              observation_class=observation_class,
              hide_cars = True if observation == "gps" else False,
              additional_sumo_cmd=f"--additional-files {net_route_files['additional']}" if observation == "camera" else None,
              sumo_warnings=False
          )
          env = pad_action_space_v0(env) # pad_action_space_v0 function pads the action space of each agent to be the same size. This is necessary for the environment to be vectorized.
          env = pad_observations_v0(env) # pad_observations_v0 function pads the observation space of each agent to be the same size. This is necessary for the environment to be vectorized.
          env = ss.pettingzoo_env_to_vec_env_v1(env) # pettingzoo_env_to_vec_env_v1 function vectorizes the PettingZoo environment for each agent, allowing it to be used with standard single-agent RL methods.
          env = ss.concat_vec_envs_v1(vec_env=env, num_vec_envs=parallelEnv, num_cpus=num_cpus, base_class="stable_baselines3") # creates parallel simulations for training
          env = VecMonitor(env)

          if mdl == 'PPO':
            model = PPO(
                "MlpPolicy",
                env=env,
                verbose=0,
                gamma=0.95,
                n_steps=256,
                ent_coef=0.0905168,
                learning_rate=0.00062211,
                vf_coef=0.042202,
                max_grad_norm=0.9,
                gae_lambda=0.99,
                n_epochs=6,
                clip_range=0.3,
                batch_size= 256,
            )
          elif mdl == 'DQN':
            model = DQN(
                env=env,
                policy="MlpPolicy",
                learning_rate=1e-3,
                batch_size= 256,
                gamma= 0.95,
```

```
            learning_starts=0,
            buffer_size=50000,
            train_freq=1,
            target_update_interval=500,
            exploration_fraction=0.05,
            exploration_final_eps=0.01,
            verbose=0,
        )
```

```
        model.learn(total_timesteps=totalTimesteps, progress_bar=True)

        model.save(model_save_path)

        env.close()
```

4. Marl simulation script

```
from stable_baselines3 import PPO
from stable_baselines3 import DQN
from stable_baselines3.common.vec_env import VecMonitor
import supersuit as ss
import sumo_rl
from supersuit.multiagent_wrappers import pad_observations_v0
from supersuit.multiagent_wrappers import pad_action_space_v0
import subprocess
from config_files.observation_class_directories import get_observation_class
from config_files.net_route_directories import get_file_locations
from config_files.delete_results import deleteSimulationResults
from config_files import reward_directories

# PARAMETERS
#=======================
# In each timestep (delta_time), the agent takes an action, and the environment (the traffic
simulation) advances by delta_time seconds.
# The agent continues to take actions for totalTimesteps.

numSeconds = 3600 # This parameter determines the total duration of the SUMO traffic simulation
in seconds.
deltaTime = 8 #This parameter determines how much time in the simulation passes with each step.
max_green = 60
simRepeats = 1 # Number of episodes
parallelEnv = 1
num_cpus = 1
maps = ["cologne1", "cologne3", "cologne8"] #['ingolstadt1', 'ingolstadt7', 'ingolstadt21']
mdl = 'PPO' # Set to DQN for DQN model
observations = ["ideal", "camera", "gps"] #camera, gps, custom
seed = '9865' # or 'random'
gui = True # Set to True to see the SUMO-GUI
yellow_time = 3 # min yellow time
add_system_info = True

#Start Simulating
if __name__ == "__main__":

    for map in maps:
        net_route_files = get_file_locations(map) # Select a map

        for observation in observations:
            #Reward
            reward_option = 'defandspeed' if observation != 'gps' else 'defandspeedwithmaxgreen'  #
'custom', 'default',
'defandmaxgreen','speed','defandspeed','defandpress','all3','avgwait','avgwaitavgspeed','defandaccum
latedspeed', 'defandmaxgreen', 'defandspeedwithmaxgreen', 'defandspeedwithphasetimes'

            # Remove results
            deleteSimulationResults(map, mdl, observation, reward_option)

            mean_reward = []

            for i in range(1, simRepeats + 1):
                # Get observation class
                observation_class = get_observation_class("model", observation)

                # Get the corresponding reward function based on the option
                reward_function = reward_directories.reward_functions.get(reward_option)

                sim_path =
f"./results/marl_sim/marl_sim-{map}-{mdl}-{observation}-{reward_option}_conn1_ep{i}"

                # creates a SUMO environment with multiple intersections, each controlled by a separate
agent.
                env = sumo_rl.parallel_env(
                    net_file=net_route_files["net"],
                    route_file=net_route_files["route"],
                    use_gui=gui,
                    num_seconds=numSeconds,
                    delta_time=deltaTime,
                    max_green=max_green,
                    out_csv_name=sim_path,
                    sumo_seed = seed,
                    yellow_time = yellow_time,
                    reward_fn=reward_function,
                    add_per_agent_info = True,
                    observation_class=observation_class,
                    hide_cars = True if observation == "gps" else False,
                    additional_sumo_cmd=f"--additional-files {net_route_files['additional']}" if
observation == "camera" else None,
                )

                env = pad_action_space_v0(env) # pad_action_space_v0 function pads the action space of
each agent to be the same size. This is necessary for the environment to be compatible with
stable-baselines3.
                env = pad_observations_v0(env) # pad_observations_v0 function pads the observation
space of each agent to be the same size. This is necessary for the environment to be compatible with
stable-baselines3.
                env = ss.pettingzoo_env_to_vec_env_v1(env) # pettingzoo_env_to_vec_env_v1 function
vectorizes the PettingZoo environment, allowing it to be used with standard single-agent RL
methods.
                env = ss.concat_vec_envs_v1(env, parallelEnv, num_cpus=num_cpus,
base_class="stable_baselines3") # function creates 4 copies of the environment and runs them in
parallel. This effectively increases the number of agents by 4 times, as each copy of the environment
has its own set of agents.
                env = VecMonitor(env)

                if mdl == 'PPO':
                    model = PPO(
                        env=env,
                        policy="MlpPolicy",
                    )
                elif mdl == 'DQN':
                    model = DQN(
                        env=env,
                        policy="MlpPolicy",
                    )

                # Run a manual simulation
                model.set_parameters(f"./models-/{map}_{mdl}_{observation}_{reward_option}",
exact_match=True, device='auto') # Set to best_model for hyper parameter tuning models
                avg_rewards = []
                obs = env.reset()
                done = False
                while not done:
                    actions = model.predict(obs, deterministic=True)[0]
                    obs, rewards, dones, infos = env.step(actions)
                    avg_rewards.append(sum(rewards)/len(rewards))
                    done = dones.any()

                print(f"\nMean reward for {map} {observation} manual simulation {i} =
{sum(avg_rewards)/len(avg_rewards)}\n")
                mean_reward.append(sum(avg_rewards)/len(avg_rewards))
                env.close()


print(f"\n=====================================================\nMean
reward for {map} {observation} all simulations=
{sum(mean_reward)/len(mean_reward)}\n=============================================
=============\n")

#Rename files
script_path = './rename.py'
folder_path = './results/marl_sim/'
```

```
subprocess.call(['python', script_path, '-f', folder_path])
```

5. Marl hyper-parameter tune script

```
import optuna
from stable_baselines3 import PPO
from stable_baselines3 import DQN
from stable_baselines3.common.vec_env import VecMonitor
import supersuit as ss
import sumo_rl
from stable_baselines3.common.evaluation import evaluate_policy
import os
from supersuit.multiagent_wrappers import pad_observations_v0
from supersuit.multiagent_wrappers import pad_action_space_v0
from config_files.observation_class_directories import get_observation_class
from config_files.net_route_directories import get_file_locations
from config_files.delete_results import deleteTuneResults
from config_files import reward_directories

# PARAMETERS
#=====================
# In each timestep (delta_time), the agent takes an action, and the environment (the traffic simulation) advances by delta_time seconds.
# The agent continues to take actions for total_timesteps.
# The policy is updated every n_steps steps, and each update involves going through the batch of interactions n_epochs times.
# The simulation repeats and improves on the previous model by repeating the simulation for a number of episodes
# This whole process is repeated for nTrials trials with different hyperparameters.

numSeconds = 3600 # This parameter determines the total duration of the SUMO traffic simulation in seconds.
deltaTime = 8 #This parameter determines how much time in the simulation passes with each step.
max_green = 60
simRepeats = 20 # Number of episodes
parallelEnv = 9
nTrials = 50
num_cpus = 4
yellow_time = 3 # min yellow time
totalTimesteps = numSeconds*simRepeats*parallelEnv # This is the total number of steps in the environment that the agent will take for training. It's the overall budget of steps that the agent can interact with the environment.
map = "cologne8"
mdl = 'PPO' # Set to DQN for DQN model
observation = "ideal" #camera, gps
reward_option = 'defandspeed'  # 'custom', 'default', 'defandmaxgreen','speed','defandspeed','defandpress','all3','avgwait','avgwaitavgspeed','defandaccumlatedspeed'
seed = '12345' # or 'random'
gui = False # Set to True to see the SUMO-GUI
add_system_info = True
net_route_files = get_file_locations(map) # Select a map
best_score = -99999

# Get observation class
observation_class = get_observation_class("model", observation)

# Get the corresponding reward function based on the option
reward_function = reward_directories.reward_functions.get(reward_option)

def runTrial(trial):

    global best_score

    print()
    print(f"Create environment for trial {trial.number}")
    print("-----------------------------------------")

    results_path = f'./results/marl_tune/marl_tune-{map}-{mdl}-{observation}-{reward_option}-trial{trial.number}'
    print(results_path)

    # creates a SUMO environment with multiple intersections, each controlled by a separate agent.
    env = sumo_rl.parallel_env(
        net_file=net_route_files["net"],
        route_file=net_route_files["route"],
        use_gui=gui,
        num_seconds=numSeconds,
        delta_time=deltaTime,
        max_green = max_green,
        out_csv_name=results_path,
        sumo_seed = seed,
        add_system_info = add_system_info,
        observation_class=observation_class,
        reward_fn=reward_function,
        add_per_agent_info = True,
        yellow_time = yellow_time,
        hide_cars = True if observation == "gps" else False,
        additional_sumo_cmd=f"--additional-files {net_route_files['additional']}" if observation == "camera" else None
    )

    env = pad_action_space_v0(env) # pad_action_space_v0 function pads the action space of each agent to be the same size. This is necessary for the environment to be compatible with stable-baselines3.
    env = pad_observations_v0(env) # pad_observations_v0 function pads the observation space of each agent to be the same size. This is necessary for the environment to be compatible with stable-baselines3.
    env = ss.pettingzoo_env_to_vec_env_v1(env) # pettingzoo_env_to_vec_env_v1 function vectorizes the PettingZoo environment, allowing it to be used with standard single-agent RL methods.
    env = ss.concat_vec_envs_v1(env, parallelEnv, num_cpus=num_cpus, base_class="stable_baselines3") # function creates 4 copies of the environment and runs them in parallel. This effectively increases the number of agents by 4 times, as each copy of the environment has its own set of agents.
    env = VecMonitor(env)

    if mdl == 'PPO':
      model = PPO(
        "MlpPolicy",
        env=env,
        verbose=3,
        gamma=0.95, # gamma=trial.suggest_float("gamma", 0.9, 0.99),
        # n_steps=256,
        n_steps=int(trial.suggest_int("n_steps", 256, 768, step=128)), # This is the number of steps of interaction (state-action pairs) that are used for each update of the policy.
        ent_coef=0.0905168, # ent_coef=trial.suggest_float("ent_coef", 0.01, 0.1),
        # learning_rate=0.00062211,
        learning_rate=trial.suggest_float("learning_rate", 1e-5, 1e-3, step=1e-5),
        vf_coef=0.042202,
        max_grad_norm=0.9,
        gae_lambda=0.99,
        # n_epochs=5,
        n_epochs=int(trial.suggest_int("n_epochs", 5, 8, step=1)),
        clip_range=0.3,
        # batch_size= 256,
        batch_size=int(trial.suggest_int("batch_size", 128, 384, step=128)),
      )
    elif mdl == 'DQN':
      model = DQN(
        env=env,
        policy="MlpPolicy",
        learning_rate=1e-3, #learning_rate=trial.suggest_float("learning_rate", 1e-5, 1e-3),
        batch_size= 256, #batch_size=int(trial.suggest_int("batch_size", 128, 512, step=128)),
        gamma= 0.95,
        learning_starts=0,
        buffer_size=50000,
        train_freq=1,
        target_update_interval=500, #update the target network every ``target_update_interval`` environment steps.
        exploration_fraction=0.05,
        exploration_final_eps=0.01,
        verbose=3,
      )

    try:

      model.learn(total_timesteps=totalTimesteps, progress_bar=True)

      # An average must be taken due to the unpredictability of the rewards
      ep_reward = []
      for i in range(1, 8):
```

```
            print(f"Testing model for {i} episode(s)")
            #Calculate the reward
            avg_rewards = []
            obs = env.reset()
            done = False
            while not done:
                actions = model.predict(obs, deterministic=True)[0]
                obs, rewards, dones, infos = env.step(actions)
                avg_rewards.append(sum(rewards)/len(rewards))
                done = dones.any()

            ep_reward.append(sum(avg_rewards)/len(avg_rewards))

        mean_reward = 0.0
        mean_reward = sum(ep_reward)/len(ep_reward)
        # mean_reward, _ = evaluate_policy(model, env, n_eval_episodes=1)
        print(f"Mean reward: {mean_reward} (params: {trial.params})")

        # Check if the current model is better than the best so far
        if mean_reward > best_score:
            best_score = mean_reward
            # Save the best model to a file
            model.save(f"./models/best_model_{map}_{mdl}_{observation}-{reward_option}")
            print("model saved")

        env.close() # Verify that this does not break the code
        return mean_reward

    except Exception as e:
        print(f"An error occurred in the current trial: {str(e)}")
        env.close()


# START TRAINING
```

```
# ======================
def objective(trial):
    final_reward = runTrial(trial)
    return final_reward


if __name__ == "__main__":

    #Delete results
    # deleteTuneResults(map, mdl, observation, reward_option)

    # Define optuna parameters
    study_name = f"multi-agent-tuned-using-optuma-{map}-{mdl}-{observation}-{reward_option}"
    storage_url =
f"sqlite:///optuna/multi-tuned-{map}-{mdl}-{observation}-{reward_option}-db.sqlite3"
    file_to_delete = f"./optuna/multi-tuned-{map}-{mdl}-{observation}-{reward_option}-db.sqlite3"

    # Check if the file exists before attempting to delete it
    if os.path.exists(file_to_delete):
        os.remove(file_to_delete)
        print(f"{file_to_delete} has been deleted.")
    else:
        print(f"{file_to_delete} does not exist in the current directory.")

    study = optuna.create_study(
        storage=storage_url,
        study_name=study_name,
        direction="maximize"
    )
    study.optimize(objective, n_trials=nTrials)
    print(f"Best value: {study.best_value} (params: {study.best_params})")
```

6. Simulating fixed control methods

```
from sumo_rl.environment.env import SumoEnvironment
from config_files.greedy.action import greedy_action
from config_files.max_pressure.action import max_pressure_action
from config_files.action_lane_relationships import get_action_lane_relationships
from config_files.net_route_directories import get_file_locations
from config_files.observation_class_directories import get_observation_class
from config_files import reward_directories
import csv

types = ["fixed", "greedy", "max_pressure"]
maps = ["beyers", "ingolstadt7", "ingolstadt21"] #choose the map to simulate
gui = False #SUMO gui
reward_option = 'defandspeed'  # 'default',
'defandmaxgreen','speed','defandspeed','defandpress','all3','avgwait','avgwaitavgspeed','defandaccum
latedspeed', 'defandmaxgreen'
num_seconds = 3600 #episode duration
delta_time = 8 #step duration
max_green = 60
yellow_time = 3 # min yellow time
simRepeats = 1
seed = "12345"

for map_name in maps:
    map = get_file_locations(map_name) #obtain network, route, and additional files
    action_lanes = get_action_lane_relationships(map_name) #dict of relationships between actions
and lanes for each intersection

    for type in types:
        observations = ["camera", "gps", "ideal"] if type != "fixed" and type != "rand" else ["none"]

        for observation in observations:

            for i in range(1, simRepeats + 1):

                print(f"Start: {map_name}-{type}-{observation}-{i}")

                # Selects the observation class specified
                observation_class = get_observation_class(type, observation)

                # Get the corresponding reward function based on the option
                reward_function = reward_directories.reward_functions.get(reward_option)
```

```
                env = SumoEnvironment(
                    net_file=map["net"],
                    route_file=map["route"],
                    use_gui=gui,
                    num_seconds=num_seconds,
                    delta_time=delta_time,
                    max_green=max_green,
                    sumo_seed=seed,
                    add_per_agent_info = True,
                    observation_class=observation_class,
                    reward_fn=reward_function,
                    yellow_time = yellow_time,
                    additional_sumo_cmd=f"--additional-files {map['additional']}" if observation ==
"camera" else None,
                    fixed_ts = True if type == "fixed" else False,
                    hide_cars = True if observation == "gps" else False
                )

                data = [] #initialize a list to store the data
                observations = env.reset()
                done = False
                avg_rewards = []
                while not done:
                    if type == "greedy":
                        actions = {agent: greedy_action(observations[agent], action_lanes[agent],
env.traffic_signals[agent].green_phase,
env.traffic_signals[agent].get_time_since_last_phase_change()[0]) for agent in env.ts_ids}
                    elif type == "max_pressure":
                        actions = {agent: max_pressure_action(observations[agent], action_lanes[agent],
env.traffic_signals[agent].green_phase,
env.traffic_signals[agent].get_time_since_last_phase_change()[0]) for agent in env.ts_ids}
                    elif type == "fixed":
                        actions = {}
                    elif type == "rand":
                        actions = {agent: env.action_spaces(agent).sample() for agent in env.ts_ids}
                    else:
                        raise ValueError(f"{type} is an invalid type for fixed control simulations")
                    observations, rewards, dones, infos = env.step(actions)
                    if type != "fixed":
                        avg_rewards.append(sum(rewards.values())/len(rewards.values()))
                    data.append(infos.copy())
                    done = dones['__all__']
```

```
        if type != "fixed":
            mean_reward = sum(avg_rewards)/len(avg_rewards)
            print(f"Mean reward for {type} {observation} simulation {i} = {mean_reward}")

        env.close()

        # Create a CSV file and write the data to it
        headings = data[0].keys()
        if data:
            with open(f"./results/{type}/{map_name}-{type}-{observation}_conn1_ep{i}.csv",
mode='w', newline='') as csv_file:
                writer = csv.DictWriter(csv_file, fieldnames=headings)
                writer.writeheader()
                writer.writerows(data)
```

7. Ranking script

```
gimport argparse
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import os

sns.set(
      style="darkgrid",
      rc={
          "figure.figsize": (7.2, 8),
          "text.usetex": False,
          "xtick.labelsize": 10,
          "ytick.labelsize": 12,
          "font.size": 15,
          "figure.autolayout": True,
          "axes.titlesize": 17,
          "axes.labelsize": 16,
          "lines.linewidth": 0.8,
          "lines.markersize": 6,
          "legend.fontsize": 8,
      },
  )

def calculate_scores(df):
    position_points = {1: 2.5, 2: 1.5, 3: 1, 4: 0.7, 5: 0.5, 6: 0.4, 7:
0.3, 8: 0.2, 9: 0.1, 10: 0}

    total_scores = {}
    system_scores = {}
    agent_scores = {}

    for _, row in df.iterrows():
        model = row['Model']
        position = row['Position']

        points = position_points.get(position, 0)

        if model not in total_scores:
            total_scores[model] = 0
        if model not in system_scores:
            system_scores[model] = 0
        if model not in agent_scores:
            agent_scores[model] = 0

        if row['Type'].startswith('system'):
            system_scores[model] += points
        elif row['Type'].startswith('agents'):
            agent_scores[model] += points
        total_scores[model] += points

    return total_scores, system_scores, agent_scores

def main():
    parser = argparse.ArgumentParser(description='Plot total scores for
models')
    parser.add_argument('-f', '--file', type=str, required=True,
help='CSV file path')
    parser.add_argument("-t", type=str, default="Title", help="Plot
title\n")
    parser.add_argument("-xh", type=str, default="Not specified", help="X
label\n")
    parser.add_argument("-xlist", type=str, default="Not specified",
help="X list sperated by , \n")
    args = parser.parse_args()
    file_path = args.file

    df = pd.read_csv(file_path)

    total_scores, system_scores, agent_scores = calculate_scores(df)
    if args.xlist == "Not specified":
        models = list(total_scores.keys())
    else :
        print(args.xlist)
        models = args.xlist.split(",")
        print(models)
    total_scores_list = list(total_scores.values())
    system_scores_list = list(system_scores.values())
    agent_scores_list = list(agent_scores.values())

    scores = {
        'Total Score': total_scores_list,
        'System Score': system_scores_list,
        'Agents Score': agent_scores_list
    }


    for score_type, score_list in scores.items():
        # Sort models and scores by model names so that each model gets
the same colour every time you run the script
        models = total_scores.keys()
        sorted_models = zip(models, score_list)
        sorted_models = sorted(sorted_models, reverse=True)
        models, score_list = zip(*sorted_models)

        # Sort models and scores based on score in descending order
        colors = plt.cm.viridis(np.linspace(0, 1, len(models)))
        sorted_lists = zip(score_list, models, colors)
        sorted_lists = sorted(sorted_lists, reverse=True)
        score_list, models, colors = zip(*sorted_lists)

        plt.figure(figsize=(8, 8))
        bars = plt.bar(models, score_list, color=colors)
        plt.xlabel(args.xh)
        plt.ylabel(score_type)
        plt.title(args.t)
        plt.grid(axis='x', linestyle='--', alpha=0.6)
        # Rotate the x-labels vertically
        plt.xticks(rotation='vertical')
        # Add total score labels on top of each bar
        for bar, score in zip(bars, score_list):
            plt.text(bar.get_x() + bar.get_width() / 2 - 0.1,
bar.get_height() + 0.05, str(round(score,2)), fontsize=10)

        # Save the plot as a PDF file
        if score_type == "Total Score":
            output_path = os.path.splitext(file_path)[0] +
"_rank_total.pdf"
        elif score_type == "System Score":
            output_path = os.path.splitext(file_path)[0] +
"_rank_system.pdf"
        else:
            output_path = os.path.splitext(file_path)[0] +
"_rank_agents.pdf"
        plt.savefig(output_path, format='pdf', bbox_inches='tight')
        print(f"Saved {output_path}")

        plt.show()

if __name__ == '__main__':
```

```
    main()
```

8. Plot training progress

```
import argparse
import glob
from itertools import cycle
from matplotlib.backends.backend_pdf import PdfPages  # Import PdfPages
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import csv

# Initialize an empty dictionary to store the sums
meanVec = {}

def setup_graphs(num):
    sns.set(
        style="darkgrid",
        rc={
            "figure.figsize": (7.2, 4.45),
            "text.usetex": False,
            "xtick.labelsize": 12,
            "ytick.labelsize": 12,
            "font.size": 15,
            "figure.autolayout": True,
            "axes.titlesize": 17,
            "axes.labelsize": 12,
            "lines.linewidth": 0.8,
            "lines.markersize": 6,
            "legend.fontsize": 8,
        },
    )
    colors = sns.color_palette("colorblind",  num)
    sns.set_palette(colors)
    colors = cycle(colors)
    return colors

dashes_styles = cycle(["-.", "-", "--"])

def compare(compVec, pdf_name):
  grouped_data = {}
  csv_column_headings = ["Type", "Score", "Model", "Position"]

  for key, value in compVec.items():
        # Extract the y-axis name from the key
        y_axis_name = key.split(",")[1]

        if y_axis_name not in grouped_data:
            grouped_data[y_axis_name] = []

        # Append the key-value pair to the corresponding group
        grouped_data[y_axis_name].append((key, value))

  with open(f"./plots/{pdf_name}.csv", mode='w', newline='') as csv_file:
        writer = csv.writer(csv_file)
        writer.writerow(csv_column_headings)

        # Sort each group by the sum (value) in ascending order
        for y_axis_name, group in grouped_data.items():
            sorted_group = sorted(group, key=lambda x: x[1], reverse=True if y_axis_name in
["system_mean_speed", "agents_mean_speed"] else False)
            pos = 1

            for key, value in sorted_group:
                model_name = key.split(",")[0]
                entry = f"{y_axis_name},{value},{model_name},{pos}"
                print(entry)
                entry = [y_axis_name, value, model_name, pos]
                writer.writerow(entry)
                pos += 1

def moving_average(interval, window_size):
    if window_size == 1:
        return interval
    window = np.ones(int(window_size)) / float(window_size)
    return np.convolve(interval, window, "same")


def plot_df(df, color, xaxis, yaxis, ma=1, label=""):
    df[yaxis] = pd.to_numeric(df[yaxis], errors="coerce")  # convert NaN string to NaN value

    mean = df.groupby(xaxis).mean()[yaxis]
    std = df.groupby(xaxis).std()[yaxis]
    if ma > 1:
        mean = moving_average(mean, ma)
        std = moving_average(std, ma)

    x = df.groupby(xaxis)[xaxis].mean().keys().values
    # plt.figure(figsize=(8, 8))
    plt.plot(x, mean, label=label, color=color, linestyle=next(dashes_styles))
    plt.fill_between(x, mean + std, mean - std, alpha=0.25, color=color, rasterized=True)


    # x = df[xaxis]
    # y = df[yaxis]
    # plt.plot(x, y, label=label, color=color, linestyle=next(dashes_styles))
    # # plt.fill_between(x, mean + std, mean - std, alpha=0.25, color=color, rasterized=True)

    # Calculate and store the sum of the second column
    sum_yaxis = df[yaxis].sum()
    count_yaxis = df[yaxis].count()
    # Create a label for the row entry in meanVec
    row_label = f"{label},{yaxis}"
    # Add the sum to the meanVec dictionary with the label as the key
    meanVec[row_label] = sum_yaxis # or avg_yaxis

def getPDFName(filenames):
  pdf_name = ""
  if len(filenames) > 1:
    parts = filenames[0].split("/")[-1].split(".")[0].split("-", 2)
    pdf_name = parts[1]
    last = filenames[0].split("/")[-1].split(".")[0].split("-", 1)[1].split("_")[1]
    groups = ""
    for filename in filenames:
      group = filename.split("/")[-1].split(".")[0].split("-", 2)[2].split("_")[0]
      groups = groups + f"-[{group}]"

    return "training-progress-" + parts[1]

  else:
    pdf_name = "training-progress-" + filenames[0].split("/")[-1].split(".")[0]
  return pdf_name

if __name__ == "__main__":

  # List of five different y-axis variables
  # y_variables = ["system_total_waiting_time", "system_accumulated_waiting_time (100)",
"system_accumulated_waiting_time (delta)", "system_total_stopped",
"system_mean_waiting_time", "system_accumulated_mean_waiting_time (100)",
"system_accumulated_mean_waiting_time (delta)", "system_mean_speed", "system_cars_present",
"agents_total_accumulated_waiting_time (100)", "agents_total_accumulated_waiting_time (delta)",
"agents_total_stopped", "agents_mean_waiting_time (100)", "agents_mean_waiting_time (delta)",
"agents_mean_speed", "agents_cars_present"]
  # y_names = ["system_total_waiting_time (s)", "system_accumulated_waiting_time (100) (s)",
"system_accumulated_waiting_time (delta) (s)","system_total_stopped (stopped vehicles)",
"system_mean_waiting_time (s)", "system_accumulated_mean_waiting_time (100) (s)",
"system_accumulated_mean_waiting_time (delta) (s)", "system_mean_speed (m/s)",
"system_cars_present", "agents_total_accumulated_waiting_time (100) (s)",
"agents_total_accumulated_waiting_time (delta) (s)", "agents_total_stopped (stopped vehicles)",
"agents_mean_waiting_time (100) (s)", "agents_mean_waiting_time (delta) (s)",
"agents_mean_speed (m/s)", "agents_cars_present"]

  y_variables = ["system_total_waiting_time", "system_accumulated_waiting_time (100)",
```

```
"system_total_stopped", "system_mean_waiting_time", "system_accumulated_mean_waiting_time
(100)", "system_mean_speed", "system_cars_present", "agents_total_accumulated_waiting_time
(100)", "agents_total_stopped", "agents_mean_waiting_time (100)", "agents_mean_speed",
"agents_cars_present"]
 y_names = ["system_total_waiting_time (s)", "system_accumulated_waiting_time (100) (s)",
"system_total_stopped (stopped vehicles)", "system_mean_waiting_time (s)",
"system_accumulated_mean_waiting_time (100) (s)", "system_mean_speed (m/s)",
"system_cars_present", "agents_total_accumulated_waiting_time (100) (s)", "agents_total_stopped
(stopped vehicles)", "agents_mean_waiting_time (100) (s)", "agents_mean_speed (m/s)",
"agents_cars_present"]

 # Create a single PDF file to save all subplots
 para = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter, description="""Plot Traffic Signal
Metrics"""
    )
 para.add_argument("-f", nargs="+", required=True, help="Measures files\n")
 para.add_argument("-conn", type=int, default=1, help="Number of conns.\n")
 para.add_argument("-start", type=int, default=1, help="Start episode.\n")
 para.add_argument("-stop", type=int, default=10, help="Stop at episode.\n")
 para.add_argument("-t", type=str, default="Title", help="Plot title\n")
 para.add_argument("-l", nargs="+", default=None, help="File's legends\n")

 pr = para.parse_args()
 filenames = pr.f
 pdf_name = getPDFName(filenames)

 try:
    pdf_filename = f"./plots/{pdf_name}.pdf"
    pdf_pages = PdfPages(pdf_filename)

 except Exception as e:
    print(f"Error: {e}")
    pdf_name = "default"
    pdf_filename = f"./plots/{pdf_name}.pdf"
    pdf_pages = PdfPages(pdf_filename)

 colors = setup_graphs(len(pr.f) if len(pr.f) > 1 else 25)


 for y_axis_variable, y_name in zip(y_variables, y_names):
    prs = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter, description="""Plot Traffic Signal
Metrics"""
    )
    prs.add_argument("-f", nargs="+", required=True, help="Measures files\n")
    prs.add_argument("-l", nargs="+", default=None, help="File's legends\n")
    prs.add_argument("-t", type=str, default=y_axis_variable, help="Plot title\n")
    prs.add_argument("-yaxis", type=str, default=y_axis_variable, help="The column to plot.\n")
    prs.add_argument("-xaxis", type=str, default="Episode", help="The x axis.\n")
    prs.add_argument("-ma", type=int, default=1, help="Moving Average Window.\n")
    prs.add_argument("-conn", type=int, default=1, help="Number of conns.\n")
    prs.add_argument("-sep", type=str, default=",", help="Values separator on file.\n")
    prs.add_argument("-xlabel", type=str, default="Time step (seconds)", help="X axis label.\n")
    prs.add_argument("-ylabel", type=str, default=y_name, help="Y axis label.\n")
    prs.add_argument("-output", type=str, default=None, help="PDF output filename.\n")
    prs.add_argument("-start", type=int, default=1, help="Start episode.\n")
    prs.add_argument("-stop", type=int, default=10, help="Stop at episode.\n")

    args = prs.parse_args()
    if args.l == None:
      labels = cycle([s.split("/")[-1] for s in args.f]) if args.f is not None else cycle([str(i) for i in
range(len(args.f))])
    else:
      labels = cycle(args.l)

    # Create a subplot for this y-axis variable
    plt.figure()
    max_y = 0
    min_y = 9999

    # Plot each model on a single graph
    for file in args.f:
       df_ep = pd.DataFrame()

       for conn in range(1, pr.conn + 1):
          # Initialize an empty list to store episode data
          episode_data = []
          ep = pr.start - 1

          try:
             # for f in glob.glob(file + str(conn) + "*"):
             for episode_num in range(pr.start, pr.stop + 1):
                try:
                   ep += 1

                   # if "marl_train" in file:
                   #     f = file + str(conn) + f"_ep{episode_num}.csv"
                   #     df = pd.read_csv(f, sep=args.sep)[["step",  y_axis_variable]]
                   # else:
                   #     raise ValueError("The word 'marl_train' is not found in the variable 'file'.")

                   f = file + str(conn) + f"_ep{episode_num}.csv"
                   df = pd.read_csv(f, sep=args.sep)[["step",  y_axis_variable]]


                except Exception as e:
                   try:
                      f = file + str(conn) + ".csv"
                      df = pd.read_csv(f, sep=args.sep)[["step",  y_axis_variable]]
                   except Exception as e:
                      ep -= 1
                      f = file + str(conn) + f"_ep{ep}.csv"
                      # df = pd.read_csv(f, sep=args.sep)[["step",  y_axis_variable]]
                      temp = []
                      values = [item[ y_axis_variable] for item in episode_data]
                      average = sum(values) / len(values)
                      temp.append({y_axis_variable: average})
                      df = pd.DataFrame.from_records(temp)

                episode_sum = df[y_axis_variable].mean()
                episode_data.append({"Episode": episode_num, y_axis_variable: episode_sum})
                if episode_sum > max_y:
                   max_y = episode_sum
                elif episode_sum < min_y:
                   min_y = episode_sum

                if df_ep.empty:
                   df_ep = pd.DataFrame.from_records(episode_data)
                else:
                   df_ep = pd.concat((df_ep, pd.DataFrame.from_records(episode_data)))

             except Exception as e:
                pass

       # Plot DataFrame
       plot_df(df_ep, xaxis=args.xaxis, yaxis=args.yaxis, label=next(labels), color=next(colors),
ma=args.ma)

       # Plot specific model f
       plt.title(args.t)
       plt.ylabel(args.ylabel)
       plt.xlabel("Episodes")
       y_range = max_y - min_y
       plt.ylim(bottom=min_y - 0.1*y_range, top=max_y + 0.1*y_range)

       # Calculate ylim values based on data
      #  min_y = df_ep[args.yaxis].min()
      #  max_y = df_ep[args.yaxis].max()
      #  y_range = max_y - min_y
      #  plt.ylim(min_y - 0.1 * y_range, max_y + 0.1 * y_range)

    # plt.show()
    plt.legend()

    # Save the current subplot to the PDF pages
    pdf_pages.savefig()

 compare(meanVec, pdf_name)  # Call the compare function to add the data to the PDF

 # Close the PDF file
 pdf_pages.close()
```

9. Rewards

```
# Custom reward functions
# =======================

def custom(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    phase_time_rwd = traffic_signal.time_since_phase_chosen_reward() #normalized by max green
times num green phases
    reward = 1*diff_wait + 50*diff_avg_speed + 0.1*phase_time_rwd
    return reward

def default(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    return diff_wait

def defandmaxgreen(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward()
    max_green_punishment = traffic_signal.max_green_reward()
    return diff_wait + max_green_punishment

def speed(traffic_signal):
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    return 5*diff_avg_speed

def pressure(traffic_signal):
    diff_pressure = traffic_signal.diff_pressure_reward()
    return 10*diff_pressure

def defandspeed(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()

    reward = 1*diff_wait + 5*diff_avg_speed

    return reward

def defandspeedwithmaxgreen(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    max_green = traffic_signal.max_green_reward()

    reward = 1*diff_wait + 5*diff_avg_speed + 0.0001*max_green

    return reward

def defandspeedwithphasetimes(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    time_since_phase = traffic_signal.time_since_phase_chosen_reward()

    reward = 1*diff_wait + 5*diff_avg_speed + time_since_phase

    return reward

def defandpress(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_pressure = traffic_signal.diff_pressure_reward()

    reward = 1*diff_wait + 0.5*diff_pressure

    return reward

def all3(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    diff_pressure = traffic_signal.diff_pressure_reward()
    reward = 1*diff_wait + 5*diff_avg_speed + 0.5*diff_pressure

    return reward

def avgwait(traffic_signal):
    diff_avg_wait = traffic_signal._diff_avg_waiting_time_reward() # average instead of
accumulated waiting time
    reward = 1*diff_avg_wait

    return reward

def avgwaitavgspeed(traffic_signal):
    diff_avg_wait = traffic_signal._diff_avg_waiting_time_reward() # average instead of
accumulated waiting time
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    reward = 1*diff_avg_wait + 0.05*diff_avg_speed

    return reward

def defandaccumlatedspeed(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_speed = traffic_signal.diff_speed_reward()

    reward = 1*diff_wait + 1*diff_speed/100

    return reward

# Create a dictionary to map reward options to functions
reward_functions = {
    'custom': custom,
    'default': default,
    'defandmaxgreen': defandmaxgreen,
    'speed': speed,
    'pressure': pressure,
    'defandspeed': defandspeed,
    'defandpress': defandpress,
    'all3': all3,
    'avgwait': all3,
    'avgwaitavgspeed': all3,
    'defandaccumlatedspeed': defandaccumlatedspeed,
    'defandspeedwithmaxgreen': defandspeedwithmaxgreen,
    'defandspeedwithphasetimes': defandspeedwithphasetimes
}
```

10. Observations

```
from sumo_rl.environment.observations import ObservationFunction
from gymnasium import spaces
from sumo_rl.environment.traffic_signal import TrafficSignal
import numpy as np

class ModelIdealObservationFunction(ObservationFunction):
    """Custom observation function for traffic signals."""

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  #
one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green +
self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles
divided by the number of vehicles that could fit in the lane.
        # time_since_last_phase_change = self.ts.get_time_since_last_phase_change()
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting
divided by the number of vehicles that could fit in the lane.
        # wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting
time per lane.
```

```
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + laneOccupancy + avgSpeedPerLane, dtype=np.float32)
        return observation

    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
        )

class GreedyIdealObservationFunction(ObservationFunction):
    def __init__(self, ts: TrafficSignal):
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the custom observation."""
        queue = self.ts.get_occupancy_per_lane()
        observation = np.array(queue, dtype=np.float32)
        return observation

    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
        return spaces.Box(
            low=np.zeros(len(self.ts.lanes), dtype=np.float32),
            high=np.ones(len(self.ts.lanes), dtype=np.float32),
        )

class MaxPressureIdealObservationFunction(ObservationFunction):
    def __init__(self, ts: TrafficSignal):
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the custom observation."""
        queue = self.ts.get_lanes_pressure()
        observation = np.array(queue, dtype=np.float32)
        return observation

    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
        return spaces.Box(
            low=np.zeros(len(self.ts.lanes), dtype=np.float32),
            high=np.ones(len(self.ts.lanes), dtype=np.float32),
        )

# For testing all other observations
class OB1(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        observation = np.array(phase_id + queue + density, dtype=np.float32)

        return observation
#
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""

        return spaces.Box(
            low=np.zeros(3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(3 * len(self.ts.lanes), dtype=np.float32),
        )

class OB2(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:

        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding

        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        observation = np.array(phase_id + queue + wait + laneOccupancy, dtype=np.float32)

        return observation

    def observation_space(self) -> spaces.Box:

        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
        )

class OB3(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:

        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        observation = np.array(phase_id + queue + wait + laneOccupancy + minDist, dtype=np.float32)

        return observation

    def observation_space(self) -> spaces.Box:

        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
        )


class OB4(ObservationFunction):
    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        # wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting
```

```
time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        # avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + laneOccupancy, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 2 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 2 * len(self.ts.lanes), dtype=np.float32),
        )
# ---------------------------------

class OB5(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        # wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        # avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + laneOccupancy + density, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 3* len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
        )
# ---------------------------------

class OB6(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + wait + laneOccupancy + density + avgSpeedPerLane, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 5 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 5 * len(self.ts.lanes), dtype=np.float32),
        )
# ---------------------------------

class OB7(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        # avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + wait + laneOccupancy + queueOut, dtype=np.float32)

        return observation
# ---------------------------------------
```

```
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
        )
# -----------------------------------

class OB8(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + wait + laneOccupancy + density + avgSpeedPerLane + minDist, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 6 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 6 * len(self.ts.lanes), dtype=np.float32),
        )
# -----------------------------------

class OB9(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + wait + laneOccupancy + density + avgSpeedPerLane + minDist + queueOut, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 7 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 7 * len(self.ts.lanes), dtype=np.float32),
        )
# -----------------------------------

class OB10(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        # wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + laneOccupancy + avgSpeedPerLane, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
        )
# -----------------------------------

class OB11(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)
```

```
    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ----------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + queue + wait + laneOccupancy + avgSpeedPerLane, dtype=np.float32)

        return observation
# ----------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ----------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
        )
# ----------------------------------

class OB12(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)
```

```
    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ----------------------------------------

        #Incoming lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encoding
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the number of vehicles that could fit in the lane.
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the number of vehicles that could fit in the lane.
        time_since_last_phase_change = self.ts.get_time_since_last_phase_change()
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersection) of each lane
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the intersection for each lane

        #Outgoing lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total number of vehicles that can fit in the outgoing lanes. This prevents the model from prioritizing phases when the cars are unable to flow through the intersection into the outgoing lanes.
        observation = np.array(phase_id + time_since_last_phase_change + queue + wait + laneOccupancy + avgSpeedPerLane, dtype=np.float32)

        return observation
# ----------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ----------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 1 + 4 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 1 +  4 * len(self.ts.lanes), dtype=np.float32),
        )
# ----------------------------------
```

11. Plot simulation results

```
import argparse
import glob
from itertools import cycle
from matplotlib.backends.backend_pdf import PdfPages  # Import PdfPages
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import csv

# Initialize an empty dictionary to store the sums
meanVec = {}

def setup_graphs(num):
    sns.set(
        style="darkgrid",
        rc={
            "figure.figsize": (7.2, 4.45),
            "text.usetex": False,
            "xtick.labelsize": 12,
            "ytick.labelsize": 12,
            "font.size": 15,
            "figure.autolayout": True,
            "axes.titlesize": 17,
            "axes.labelsize": 12,
            "lines.linewidth": 0.8,
            "lines.markersize": 6,
            "legend.fontsize": 8,
        },
    )
    colors = sns.color_palette("colorblind",  num)
    sns.set_palette(colors)
    colors = cycle(colors)
    return colors

dashes_styles = cycle(["-.", "-", "--"])

def compare(compVec, pdf_name):
  grouped_data = {}
  csv_column_headings = ["Type", "Score", "Model", "Position"]

  for key, value in compVec.items():
        # Extract the y-axis name from the key
        y_axis_name = key.split(",")[1]

        if y_axis_name not in grouped_data:
            grouped_data[y_axis_name] = []

        # Append the key-value pair to the corresponding group
        grouped_data[y_axis_name].append((key, value))

  with open(f"./plots/{pdf_name}.csv", mode='w', newline='') as csv_file:
        writer = csv.writer(csv_file)
        writer.writerow(csv_column_headings)
```

```
        # Sort each group by the sum (value) in ascending order
        for y_axis_name, group in grouped_data.items():
            sorted_group = sorted(group, key=lambda x: x[1], reverse=True if y_axis_name in
["system_mean_speed", "agents_mean_speed"] else False)
            pos = 1

            for key, value in sorted_group:
                model_name = key.split(",")[0]
                entry = f"{y_axis_name},{value},{model_name},{pos}"
                print(entry)
                entry = [y_axis_name, value, model_name, pos]
                writer.writerow(entry)
                pos += 1

def moving_average(interval, window_size):
    if window_size == 1:
        return interval
    window = np.ones(int(window_size)) / float(window_size)
    return np.convolve(interval, window, "same")


def plot_df(df, color, xaxis, yaxis, ma=1, label=""):
    df[yaxis] = pd.to_numeric(df[yaxis], errors="coerce")  # convert NaN string to NaN value

    mean = df.groupby(xaxis).mean()[yaxis]
    std = df.groupby(xaxis).std()[yaxis]
    if ma > 1:
        mean = moving_average(mean, ma)
        std = moving_average(std, ma)

    x = df.groupby(xaxis)[xaxis].mean().keys().values
    plt.plot(x, mean, label=label, color=color, linestyle=next(dashes_styles))
    plt.fill_between(x, mean + std, mean - std, alpha=0.25, color=color, rasterized=True)

    # Calculate and store the sum of the second column
    sum_yaxis = df[yaxis].sum()
    count_yaxis = df[yaxis].count()
    # Calculate the average (avg_yaxis) by dividing sum_yaxis by count_yaxis
    avg_yaxis = sum_yaxis / count_yaxis
    # Create a label for the row entry in meanVec
    row_label = f"{label},{yaxis}"
    # Add the sum to the meanVec dictionary with the label as the key
    meanVec[row_label] = avg_yaxis # or avg_yaxis

def getPDFName(filenames):
  pdf_name = ""
  if len(filenames) > 1:
    parts = filenames[0].split("/")[-1].split(".")[0].split("-", 2)
    pdf_name = parts[1]
    last = filenames[0].split("/")[-1].split(".")[0].split("-", 1)[1].split("_")[1]
    groups = ""
    for filename in filenames:
      group = filename.split("/")[-1].split(".")[0].split("-", 2)[2].split("_")[0]
      groups = groups + f"-[{group}]"

    return parts[1] + "-" + parts[0]

  else:
    pdf_name = filenames[0].split("/")[-1].split(".")[0]
  return pdf_name

if __name__ == "__main__":

  # List of five different y-axis variables
  # y_variables = ["system_total_waiting_time", "system_accumulated_waiting_time (100)",
"system_accumulated_waiting_time (delta)", "system_total_stopped",
"system_mean_waiting_time", "system_accumulated_mean_waiting_time (100)",
"system_accumulated_mean_waiting_time (delta)", "system_mean_speed", "system_cars_present",
"agents_total_accumulated_waiting_time (100)", "agents_total_accumulated_waiting_time (delta)",
"agents_total_stopped", "agents_mean_waiting_time (100)", "agents_mean_waiting_time (delta)",
"agents_mean_speed", "agents_cars_present"]
  # y_names = ["system_total_waiting_time (s)", "system_accumulated_waiting_time (100) (s)",
"system_accumulated_waiting_time (delta) (s)","system_total_stopped (stopped vehicles)",
"system_mean_waiting_time (s)", "system_accumulated_mean_waiting_time (100) (s)",
"system_accumulated_mean_waiting_time (delta) (s)", "system_mean_speed (m/s)",
"system_cars_present", "agents_total_accumulated_waiting_time (100) (s)",
"agents_total_accumulated_waiting_time (delta) (s)", "agents_total_stopped (stopped vehicles)",
"agents_mean_waiting_time (100) (s)", "agents_mean_waiting_time (delta) (s)",
"agents_mean_speed (m/s)", "agents_cars_present"]
  y_variables = ["system_total_waiting_time", "system_accumulated_waiting_time (100)",
"system_total_stopped", "system_mean_waiting_time", "system_accumulated_mean_waiting_time
(100)", "system_mean_speed", "system_cars_present", "agents_total_accumulated_waiting_time
(100)", "agents_total_stopped", "agents_mean_waiting_time (100)", "agents_mean_speed",
"agents_cars_present"]
  y_names = ["system_total_waiting_time (s)", "system_accumulated_waiting_time (100) (s)",
"system_total_stopped (stopped vehicles)", "system_mean_waiting_time (s)",
"system_accumulated_mean_waiting_time (100) (s)", "system_mean_speed (m/s)",
"system_cars_present", "agents_total_accumulated_waiting_time (100) (s)", "agents_total_stopped
(stopped vehicles)", "agents_mean_waiting_time (100) (s)", "agents_mean_speed (m/s)",
"agents_cars_present"]

  # Create a single PDF file to save all subplots
  para = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter, description="""Plot Traffic Signal
Metrics"""
      )
  para.add_argument("-f", nargs="+", required=True, help="Measures files\n")
  para.add_argument("-l", nargs="+", default=None, help="File's legends\n")
  para.add_argument("-t", type=str, default="Title", help="Plot title\n")
  pr = para.parse_args()
  filenames = pr.f
  pdf_name = getPDFName(filenames)

  try:
      pdf_filename = f"./plots/{pdf_name}.pdf"
      pdf_pages = PdfPages(pdf_filename)

  except Exception as e:
      print(f"Error: {e}")
      pdf_name = "default"
      pdf_filename = f"./plots/{pdf_name}.pdf"
      pdf_pages = PdfPages(pdf_filename)

  colors = setup_graphs(len(pr.f) if len(pr.f) > 1 else 25)

  for y_axis_variable, y_name in zip(y_variables, y_names):
      prs = argparse.ArgumentParser(
          formatter_class=argparse.ArgumentDefaultsHelpFormatter, description="""Plot Traffic Signal
Metrics"""
      )
      prs.add_argument("-f", nargs="+", required=True, help="Measures files\n")
      prs.add_argument("-l", nargs="+", default=None, help="File's legends\n")
      prs.add_argument("-t", type=str, default=y_axis_variable, help="Plot title\n")
      prs.add_argument("-yaxis", type=str, default=y_axis_variable, help="The column to plot.\n")
      prs.add_argument("-xaxis", type=str, default="step", help="The x axis.\n")
      prs.add_argument("-ma", type=int, default=1, help="Moving Average Window.\n")
      prs.add_argument("-sep", type=str, default=",", help="Values separator on file.\n")
      prs.add_argument("-xlabel", type=str, default="Time step (seconds)", help="X axis label.\n")
      prs.add_argument("-ylabel", type=str, default=y_name, help="Y axis label.\n")
      prs.add_argument("-output", type=str, default=None, help="PDF output filename.\n")

      args = prs.parse_args()
      if args.l == None:
        labels = cycle([s.split("/")[-1] for s in args.f]) if args.f is not None else cycle([str(i) for i in
range(len(args.f))])
      else:
        labels = cycle(args.l)

      # Create a subplot for this y-axis variable
      plt.figure()

      # File reading and grouping
      for file in args.f:
          main_df = pd.DataFrame()
          for f in glob.glob(file + "*"):
              df = pd.read_csv(f, sep=args.sep)[["step",  y_axis_variable]]
              if main_df.empty:
                  main_df = df
              else:
                  main_df = pd.concat((main_df, df))

          # Plot DataFrame
          plot_df(main_df, xaxis=args.xaxis, yaxis=args.yaxis, label=next(labels), color=next(colors),
ma=args.ma)

          plt.title(args.t)
          plt.ylabel(args.ylabel)
          plt.xlabel(args.xlabel)
          plt.ylim(bottom=0)
```

```
    plt.legend()

    # Save the current subplot to the PDF pages
    pdf_pages.savefig()

compare(meanVec, pdf_name)  # Call the compare function to add the data to the PDF

# Close the PDF file
pdf_pages.close()
```

APPENDIX I: OBSERVATION SPACE

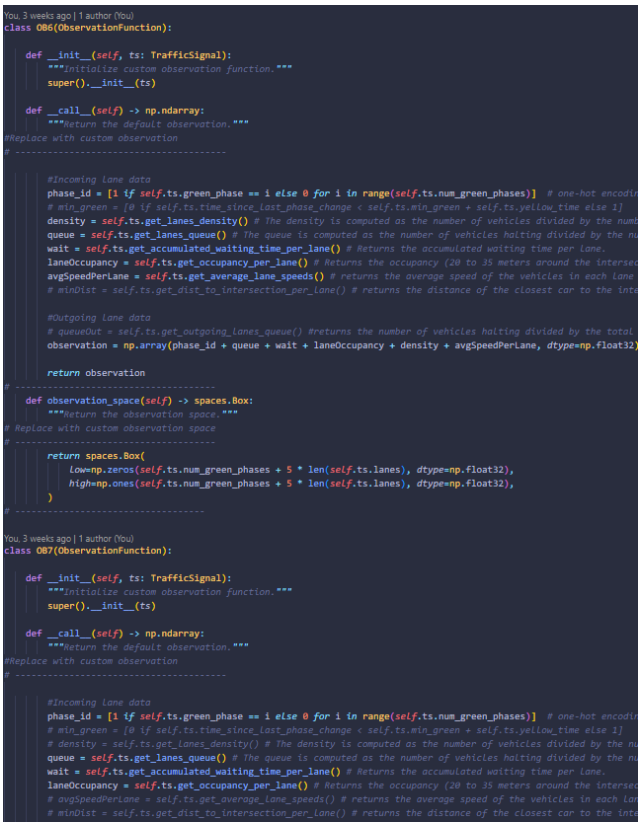
1. General:
    a. The current phase of the traffic light (one hot encoded)
    b. A parameter indicating whether the time has passed for the minimum green time of the current phase. (If it has not passed the action the model makes will not affect the simulation).
    c. Accumulated waiting time per lane
    d. Distance of head car from intersection per lane
    e. Queues of outgoing lanes
    f. Average speed of each lane
    g. Occupancy within 35m of each lane
2. Using monitoring cameras:
    a. Lane densities within camera detection region (cars seen divided by total capacity of region)
    b. Lane pressures within camera detection region (cars incoming minus cars outgoing)
3. Using google maps representation (only certain cars are visible):
    a. Lane densities within entire incoming road
    b. Lane queues (vehicles with speed less than 0.1m/s divided by total capacity of road) within entire incoming road
    c. Average speed of visible cars per lane
    d. Accumulated waiting time of visible cars per lane
    e. A list of times since each phase was chosen
    f. Occupancy of visible cars within 35m per lane

```python
from sumo_rl.environment.observations import ObservationFunction
from gymnasium import spaces
from sumo_rl.environment.traffic_signal import TrafficSignal
import numpy as np

Michael Rolle, 3 weeks ago | 2 authors (You and others)
class ModelIdealObservationFunction(ObservationFunction):
    """Custom observation function for traffic signals."""

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
        #Incoming Lane data        You, 4 weeks ago • additional observations
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divide
        # time_since_last_phase_change = self.ts.get_time_since_last_phase_change()
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divide
        # wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time pe
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around t
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car

        #Outgoing Lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by
        observation = np.array(phase_id + queue + laneOccupancy + avgSpeedPerLane, dtype=np.float32)
        return observation

    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
        )

You, 3 weeks ago | 1 author (You)
class GreedyIdealObservationFunction(ObservationFunction):
    def __init__(self, ts: TrafficSignal):
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the custom observation."""
        queue = self.ts.get_occupancy_per_lane()
        observation = np.array(queue, dtype=np.float32)
        return observation

    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
        return spaces.Box(
            low=np.zeros(len(self.ts.lanes), dtype=np.float32),
            high=np.ones(len(self.ts.lanes), dtype=np.float32),
        )
```

```python
You, 4 weeks ago | 1 author (You)
class MaxPressureIdealObservationFunction(ObservationFunction):
    def __init__(self, ts: TrafficSignal):
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the custom observation."""
        queue = self.ts.get_lanes_pressure()
        observation = np.array(queue, dtype=np.float32)
        return observation

    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
        return spaces.Box(
            low=np.zeros(len(self.ts.lanes), dtype=np.float32),
            high=np.ones(len(self.ts.lanes), dtype=np.float32),
        )

# For testing all other observations
You, 3 weeks ago | 1 author (You)
class OB1(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
        #Incoming Lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  #
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divi
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting d
        observation = np.array(phase_id + queue + density, dtype=np.float32)

        return observation
#
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""

        return spaces.Box(
            low=np.zeros(3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(3 * len(self.ts.lanes), dtype=np.float32),
        )
```

```python
You, 3 weeks ago | 1 author (You)
class OB2(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:

        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]

        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters ar
        observation = np.array(phase_id + queue + wait + laneOccupancy, dtype=np.float32)

        return observation

    def observation_space(self) -> spaces.Box:

        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 3 * len(self.ts.lanes), dtype=np.float32),
        )

You, 3 weeks ago | 1 author (You)
class OB3(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:

        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters ar
        minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest

        observation = np.array(phase_id + queue + wait + laneOccupancy + minDist, dtype=np.float32)

        return observation

    def observation_space(self) -> spaces.Box:

        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 4 * len(self.ts.lanes), dtype=np.float32),
        )
```

```python
class OB4(ObservationFunction):
    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:

        #Incoming Lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_ti
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divi
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divi
        # wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time p
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around
        # avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car

        #Outgoing Lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided t
        observation = np.array(phase_id + queue + laneOccupancy, dtype=np.float32)

        return observation
# ------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 2 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 2 * len(self.ts.lanes), dtype=np.float32),
        )
# ------------------------------------

You, 3 weeks ago | 1 author (You)
class OB5(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ------------------------------------

        #Incoming Lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one
```



```
You, 3 weeks ago | 1 author (You)
class OB6(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------


        #Incoming Lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encodi
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the numb
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the nu
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersec
        avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lane
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the inte

        #Outgoing Lane data
        # queueOut = self.ts.get_outgoing_lanes_queue() #returns the number of vehicles halting divided by the total
        observation = np.array(phase_id + queue + wait + laneOccupancy + density + avgSpeedPerLane, dtype=np.float32)

        return observation
# ---------------------------------------
    def observation_space(self) -> spaces.Box:
        """Return the observation space."""
# Replace with custom observation space
# ---------------------------------------
        return spaces.Box(
            low=np.zeros(self.ts.num_green_phases + 5 * len(self.ts.lanes), dtype=np.float32),
            high=np.ones(self.ts.num_green_phases + 5 * len(self.ts.lanes), dtype=np.float32),
        )
# ---------------------------------
You, 3 weeks ago | 1 author (You)
class OB7(ObservationFunction):

    def __init__(self, ts: TrafficSignal):
        """Initialize custom observation function."""
        super().__init__(ts)

    def __call__(self) -> np.ndarray:
        """Return the default observation."""
#Replace with custom observation
# ---------------------------------------


        #Incoming Lane data
        phase_id = [1 if self.ts.green_phase == i else 0 for i in range(self.ts.num_green_phases)]  # one-hot encodi
        # min_green = [0 if self.ts.time_since_last_phase_change < self.ts.min_green + self.ts.yellow_time else 1]
        # density = self.ts.get_lanes_density() # The density is computed as the number of vehicles divided by the nu
        queue = self.ts.get_lanes_queue() # The queue is computed as the number of vehicles halting divided by the nu
        wait = self.ts.get_accumulated_waiting_time_per_lane() # Returns the accumulated waiting time per lane.
        laneOccupancy = self.ts.get_occupancy_per_lane() # Returns the occupancy (20 to 35 meters around the intersec
        # avgSpeedPerLane = self.ts.get_average_lane_speeds() # returns the average speed of the vehicles in each lan
        # minDist = self.ts.get_dist_to_intersection_per_lane() # returns the distance of the closest car to the inte
```

APPENDIX J: REWARD FUNCTIONS

1. diff_waiting_time_reward(): Difference in the accumulated wait time over subsequent actions normalized
2. diff_avg_speed_reward(): Difference in the average speed over subsequent actions normalized via the speed limit
3. diff_speed_reward(): Difference in the accumulated speed over subsequent actions normalized via the speed limit
4. max_green_reward(): Punish the model if the agent hasn't changed its phase in max_green seconds
5. reward_highest_occupancy_phase(): Reward the model for choosing a phase for lanes with the highest occupancy
6. diff_pressure_reward(): Compute the difference in pressure between the current and the previous time step normalized by the maximum number of vehicles that can fit in that lane.

```python
# Custom reward functions
# ======================

def custom(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    phase_time_rwd = traffic_signal.time_since_phase_chosen_reward() #normalize
    reward = 1*diff_wait + 50*diff_avg_speed + 0.1*phase_time_rwd
    return reward

def default(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    return diff_wait

def defandmaxgreen(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward()
    max_green_punishment = traffic_signal.max_green_reward()
    return diff_wait + max_green_punishment

def speed(traffic_signal):
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    return 5*diff_avg_speed

def pressure(traffic_signal):
    diff_pressure = traffic_signal.diff_pressure_reward()
    return 10*diff_pressure

def defandspeed(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()

    reward = 1*diff_wait + 5*diff_avg_speed

    return reward

def defandspeedwithmaxgreen(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    max_green = traffic_signal.max_green_reward()

    reward = 1*diff_wait + 5*diff_avg_speed + 0.0001*max_green

    return reward

def defandspeedwithphasetimes(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    time_since_phase = traffic_signal.time_since_phase_chosen_reward()

    reward = 1*diff_wait + 5*diff_avg_speed + time_since_phase

    return reward

def defandpress(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_pressure = traffic_signal.diff_pressure_reward()

    reward = 1*diff_wait + 0.5*diff_pressure

    return reward
```

```python
def all3(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    diff_pressure = traffic_signal.diff_pressure_reward()
    reward = 1*diff_wait + 5*diff_avg_speed + 0.5*diff_pressure

    return reward

def avgwait(traffic_signal):
    diff_avg_wait = traffic_signal._diff_avg_waiting_time_reward() # average
    reward = 1*diff_avg_wait

    return reward

def avgwaitavgspeed(traffic_signal):
    diff_avg_wait = traffic_signal._diff_avg_waiting_time_reward() # average
    diff_avg_speed = traffic_signal.diff_avg_speed_reward()
    reward = 1*diff_avg_wait + 0.05*diff_avg_speed

    return reward

def defandaccumlatedspeed(traffic_signal):
    diff_wait = traffic_signal._diff_waiting_time_reward() # Default reward
    diff_speed = traffic_signal.diff_speed_reward()

    reward = 1*diff_wait + 1*diff_speed/100

    return reward

# Create a dictionary to map reward options to functions
reward_functions = {
    'custom': custom,
    'default': default,
    'defandmaxgreen': defandmaxgreen,
    'speed': speed,
    'pressure': pressure,
    'defandspeed': defandspeed,
    'defandpress': defandpress,
    'all3': all3,
    'avgwait': all3,
    'avgwaitavgspeed': all3,
    'defandaccumlatedspeed': defandaccumlatedspeed,
    'defandspeedwithmaxgreen': defandspeedwithmaxgreen,
    'defandspeedwithphasetimes': defandspeedwithphasetimes
}
```

Figure1: List of reward functions