FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

DynamiCITY Sim: simulation-based network assignment for sustainable cities

Luís Rafael Fernandes Mendes Afonso

Mestrado em Engenharia Informática e Computação

Supervisor: Ana Paula Rocha

Second Supervisor: António Costa

February 16, 2023

DynamiCITY Sim: simulation-based network assignment for sustainable cities

Luís Rafael Fernandes Mendes Afonso

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Rui Camacho
Referee: Prof. João Emílio Almeida
Referee: Prof. Ana Paula Rocha

February 16, 2023

This work is a result of project DynamiCITY: Fostering Dynamic Adaptation of Smart Cities to Cope with Crises and Disruptions, with reference NORTE-01-0145-FEDER-000073, supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF).

Cofinanciado por:

Resumo

Motores de desenvolvimento económico, as cidades têm crescido tanto em extensão como população, no entanto são também fortes impulsionadoras da degradação ambiental. Isto levou as Nações Unidas a incluir o transporte sustentável como um dos alvos para a Agenda 2030 para o desenvolvimento sustentável.

Uma abordagem para enfrentar este desafio é melhorar a eficiência das estruturas rodoviárias, componentes centrais de todas as cidades. Neste contexto, as ferramentas de simulação de tráfego são recursos valiosos ao ajudarem investigadores a: analisar o estado atual dos sistemas; identificar possíveis causas e soluções para problemas no fluxo de trânsito; prever os efeitos de diversas alterações à rede rodoviária. Em geral, estas ferramentas oferecem formas rápidas e baratas de planear melhorias na cidade minimizando o incómodo à população.

Os simuladores mais estabelecidos tendem para uma de duas categorias. A microscópica, que procura oferecer uma representação muito detalhada do mundo e das complexas interações entre múltiplos veículos, ou a macroscópica, que abstrai o conceito de veículos e opera sobre medidas agregadoras para gerar uma visão completa das dinâmicas do trânsito como um todo. As limitações relativas à escala de simulação exibidas pelos modelos microscópicos e a falta de detalhe nos modelos macroscópicos têm levado a um aumento no interesse por ferramentas que combinam os pontos fortes destes dois modelos e é comummente referido como o modelo mesoscópico.

Propomos o DynamiCITY Sim, um simulador escalável e que opera nesta escala intermédia, estabelecendo um equilíbrio entre dimensão e detalhe. Adicionalmente, mostramos cenários de aprendizagem por reforço multi-agente combinando o nosso simulador com RLlib, uma biblioteca open-source para aprendizagem por reforço e um ambiente OpenAI Gym personalizado.

Validamos o DynamiCITY Sim testando com o fluxo de trânsito esperado para a cidade do Porto num período de 24 horas. Quanto ao sistema de aprendizagem, validamos confirmando primeiro que os agentes são capazes de encontrar o caminho mais curto e depois observando os seus resultados num problema de afectação de tráfego, a rede Sioux Falls que possui uma solução ótima conhecida.

Os resultados obtidos mostram que o simulador consegue resolver cenários com milhões de veículos, o que confortavelmente ultrapassa a procura esperada para a cidade do Porto. Adicionalmente, a abordagem com MARL mostrou-se capaz de capturar processos de tomada de decisão dos condutores, nomeadamente, quando confrontados com quantidades elevadas de trânsito, os agentes abdicam de caminhos mais curtos em prol de caminhos menos obstruídos.

Palavras-chave: Simulação, Modelação, Multiagente, Aprendizagem por reforço, MARL, Mobilidade urbana

Abstract

Powerhouses of economic growth, cities have become not only increasingly larger and more populous but also strong drivers of environmental degradation. This led the United Nations to include sustainable transport as a target in their proposed 2030 Agenda for Sustainable Development.

One approach to tackle this objective is to improve the efficiency of road network systems, a core component of every city. In this context, traffic simulations are valuable tools, as they allow researchers to: analyse the current state of the system; identify the causes and possible solutions to traffic congestion problems; predict the effects of alterations to the road network. Overall, these tools allow for fast and inexpensive ways to plan improvements in a city while minimising disturbances to the population.

Current staple simulators often fall into one of two categories. Microscopic, which aims to deliver a very detailed representation of the environment, especially the intricacies behind the interactions between different vehicles, or macroscopic, which abstracts the concept of vehicles and deals in aggregate measurements to provide a bird's eye view of the traffic dynamics as a whole. The limitations in scale exhibited by microscopic models and the lack of detail in macroscopic models have led to increased interest in tools combining the strengths of both methods, often referred to as the mesoscopic scale.

We propose DynamiCITY Sim, a scalable simulator that operates at the mesoscopic scale, striking a balance between detail and scope. Additionally, we showcase multi-agent reinforcement learning (MARL) scenarios by combining our simulator with RLlib, an open-source library for reinforcement learning, and a custom OpenAI Gym environment.

We validate DynamiCITY Sim by testing the expected traffic demand for Porto in a 24-hour period. We also integrate our simulator into a machine learning system to: confirm if agents can find the shortest path; train them to solve a traffic assignment problem with a known optimal solution.

Our results show that our simulator can resolve scenarios with millions of drivers, far surpassing the demand for Porto. In addition, our approach to MARL was able to capture expected behaviours of drivers when faced with increasing amounts of traffic, such as opting for longer but less congested routes.

Keywords: Simulation, Modelling, Multi-Agent, Reinforcement Learning, MARL, Urban Mobility

Agradecimentos

Obrigado.

Obrigado a todos os que me ajudaram a chegar até aqui. Cheguei sobre ombros de gigantes.

Obrigado Pais, Avós e Tios pelo vosso constante apoio e confiança em mim, especialmente durante os meus momentos mais indecisos.

Obrigado Dani, obrigado por teres estado sempre presente e pronta para ajudar, fosse no que fosse. Obrigado por me ouvires pensar em voz alta horas a fio e por sempre me conseguires fazer ver que afinal está tudo bem.

Obrigado aos Amigos com quem partilhei esta jornada e que a rechearam de momentos de diversão e companheirismo.

Obrigado aos Professores que me propuseram e guiaram neste desafio.

Luís Rafael Afonso

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives . . . 2
- 1.4 Document Structure . . . 2

2 Background Knowledge **4**
- 2.1 Traffic simulation . . . 4
- 2.2 Game Theory . . . 6
- 2.3 Reinforcement Learning . . . 7
  - 2.3.1 Multi-agent Reinforcement Learning . . . 9
  - 2.3.2 Popular Reinforcement Learning Algorithms . . . 10
- 2.4 Summary . . . 10

3 Related Work **11**
- 3.1 Multi-Agent Reinforcement Learning . . . 11
  - 3.1.1 Usage in traffic context . . . 11
  - 3.1.2 Relevant MARL frameworks . . . 12
- 3.2 Traffic Simulation Models . . . 13
- 3.3 Game Theory in Traffic . . . 14
- 3.4 Gap analysis . . . 15
- 3.5 Summary . . . 15

4 DynamiCITY Sim: simulation-based network assignment for sustainable cities **16**
- 4.1 Problem Statement . . . 16
- 4.2 Proposed Solution . . . 17
- 4.3 Validation . . . 18
  - 4.3.1 Simulator Experiments . . . 18
  - 4.3.2 Reinforcement Learning Experiments . . . 19
- 4.4 Summary . . . 21

5 DynamiCITY Sim **22**
- 5.1 Mesoscopic Model . . . 22
- 5.2 Event Driven . . . 23
- 5.3 Software Description . . . 24
  - 5.3.1 Installation . . . 24
  - 5.3.2 Inputs . . . 24
  - 5.3.3 Outputs . . . 28
- 5.4 Requirements . . . 29
- 5.5 Implementation . . . 29
  - 5.5.1 Driver . . . 29
  - 5.5.2 Network Model . . . 29
  - 5.5.3 Simulator . . . 31
- 5.6 Summary . . . 31

6 DynamiCITY Learn **32**
6.1 Single-agent Environment . . . 32
6.1.1 Defining the environment . . . 32
6.1.2 Interacting with the environment . . . 33
6.2 Multi-agent Environment . . . 34
6.3 Training Process . . . 37
6.3.1 Setup . . . 37
6.3.2 Training . . . 38
6.3.3 Data Collection and Visualization . . . 39
6.4 Software Description . . . 40
6.4.1 Installation . . . 40
6.4.2 Dependencies . . . 40
6.5 Summary . . . 41

7 Results and Discussion **42**
7.1 Simulator Experiments . . . 42
7.2 Reinforcement Learning Experiments . . . 46
7.3 Summary . . . 52

8 Conclusions and Future Work **53**
8.1 Main Contributions . . . 54
8.2 Future Work . . . 54

References **56**

List of Figures

2.1 Distinction between simulation types [54] . . . 4
2.2 Microscopic, mesoscopic and macroscopic simulation model example[67] . . . 5
2.3 Interaction between agent and environment in reinforcement learning [59] . . . 8
2.4 Multiple agents in the same environment [42] . . . 9

4.1 Independent simulator usage . . . 17
4.2 Interactions between the two modules . . . 18
4.3 Network of the city of Porto . . . 19
4.4 Simple validation network . . . 19
4.5 Anaheim network[19] . . . 20
4.6 Sioux Falls network[19] . . . 20

5.1 Example of a network graph . . . 22
5.2 Computation instances in event-driven and fixed timestep simulations . . . 23
5.3 Help menu . . . 24
5.4 Snippet from an OD matrix file . . . 28
5.5 Detailed output . . . 28
5.6 Simplified output . . . 28
5.7 Example of an edge updating its flow . . . 30

6.1 Example network . . . 33
6.2 Possible state observation . . . 33
6.3 Training flow chart . . . 36
6.4 Output throughout training . . . 38
6.5 Snapshot of Sioux-Falls network with Dijkstra's shortest path assignment . . . 39

7.1 Execution time of routes with 10 steps . . . 42
7.2 Execution time of 10 drivers . . . 43
7.3 Memory consumption for both scenarios . . . 44
7.4 Simulation time when increasing the number of vehicles and steps . . . 44
7.5 Traffic prediction for Porto at 8 am . . . 45
7.6 Traffic prediction for Porto at 9 am . . . 46
7.7 Cumulative reward obtained in saturation test with PPO and A2C . . . 47
7.8 Path discovery success rate for Sioux Falls, Anaheim and Porto networks . . . 48
7.9 Route evaluation for Sioux-Falls, Anaheim and Porto networks . . . 48
7.10 Shortest path assignment . . . 50
7.11 Best known solution . . . 50
7.12 PPO exploratory behaviour . . . 50
7.13 PPO final result . . . 50

List of Tables

3.1 MARL frameworks 12
3.2 Simulator comparison 14

7.1 24-hour simulation for the city of Porto 45

Listings

5.1 Network XML Schema . . . 25
5.2 Example of a network file . . . 26
5.3 Example of a routes file . . . 27
5.4 Example of a TAZ file . . . 27
6.1 Example of a configuration dictionary for PPO . . . 37
6.2 Creating and running a RLlib Trainer . . . 38
6.3 Loading a model and performing one episode . . . 39

Abbreviations

A2C	Advantage Actor-Critic
BPR	Bureau of Public Roads
CPU	Central Processing Unit
GT	Game Theory
GPU	Graphics Processing Unit
MARL	Multi-Agent Reinforcement Learning
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
OD	Origin-destination
VDF	Volume-Delay Functions
TAZ	Traffic Assignment Zones

Chapter 1

Introduction

1.1 Context

Urban mobility is a crucial piece of sustainable development. It encompasses everything required for people to get from one place to another, be it work or leisure. As cities grow in size and population, the need for an efficient array of transportation options becomes more evident.

Urban mobility is becoming increasingly important considering its impact on CO_2 emissions and, therefore, global warming, which is why it is now seen as one of the sustainable development goals preconized by the United Nations [1].

As such, the results of measures taken to improve these systems could mean a significant reduction in harmful emissions, an overall increase in public health and a noticeable improvement in everyday commuters' quality of life. This means fewer traffic jams, less time spent waiting for public transport and an overall increase in well-being while commuting.

This work was conducted in collaboration with LIACC, the Artificial Intelligence and Computer Science Laboratory from the Faculty of Engineering of the University of Porto. It is also integrated into the DynamiCITY [2] project, which aims to develop a virtual laboratory to engineer, implement and deploy solutions for smart cities by bringing together researchers, service providers and software developers.

1.2 Motivation

Traffic flow constitutes a significant portion of the total percentage of greenhouse gas emissions [44, 18]. As such, it is vital to make substantial efforts to improve the performance of this core system of today's world.

Making changes to a city's road network is quite expensive and lengthy. Simulations offer a cheap and quick alternative to make preliminary tests and prototypes before changing the city. For

[1] https://sdgs.un.org/goals (Last Access: January 2023)

[2] DynamiCITY: Fostering Dynamic Adaptation of Smart Cities to Cope with Crises and Disruptions, P2020|Norte2020-Projetos Integrados ICDT

these experiments to be meaningful, we need tools that accurately model the life of a city in the real world.

A scalable simulator capable of accurately capturing the intricacies of the competition for resources in a traffic flow environment is an excellent tool for researchers to design and prototype modern solutions for urban mobility. Many simulators available are geared towards representing traffic at the most intricate level, which inevitably leads to scalability issues, mainly when dealing with a city-wide scale. Others describe the global dynamics of traffic flow at a large scale by treating traffic as an aggregate measure, which could lead to discrepancies regarding the interactions between individual vehicles.

With a mesoscopic simulator, we can strike a balance between scale and detail, taking advantage of the strengths of both micro and macroscopic models that better suit our needs. Additionally, we can incorporate MARL to capture the choices that drivers have to make when sharing a road network.

Although many simulators already target the mesoscopic scope [1, 5, 6, 8, 20, 14, 15, 57], there is still much to contribute at this scale, particularly when combining it with MARL.

1.3 Objectives

We strive to create a tool that enables researchers to conduct traffic experiments in a digital rendition of a city by fulfilling the following objectives:

Firstly, the work encompasses the development of a scalable event-driven simulator that operates at the mesoscopic scale. This means that we will favour the usage of a less detailed representation by using aggregate measures instead of a physics-based approach but keeping a clear distinction between drivers.

Lastly, the simulator serves as a playground to implement a multi-agent reinforced learning approach to observe the decision-making mechanisms that arise when multiple agents try to maximize resource allocation, which in this case consists of drivers competing to complete their intended routes in the least amount of time possible.

1.4 Document Structure

In this chapter, we delivered a general overview of this thesis by introducing the context in which it is inserted, its motivation and its objectives.

Chapter 2 is dedicated to providing a brief theoretical basis on some of the concepts essential to our work.

In chapter 3, we showcase some parallel works identified, namely recent applications of Multi-agent Reinforcement Learning applied to traffic. We also report on a different approach to the study of traffic with the usage of game theory. Finally, there is a review of current tools for modelling and simulating traffic and frameworks available for multi-agent reinforcement learning.

Chapter 4 embodies our problem statement, the general outlines of the proposed solution and a rundown of the resources used for testing and validating the tool developed.

Chapters 5 and 6 contain a collection of details regarding the implementation of the simulator and the MARL components, respectively. As such, they explain how these two blocks were developed and provide relevant instructions on how to operate them.

Chapter 7 entails the discussion of the results obtained from both the simulator and machine learning experiments that we have conducted and some insights collected throughout this process.

Finally, Chapter 8 closes this thesis with a review of the work done and an analysis of the different avenues of future work that might stem from this thesis.

Chapter 2

Background Knowledge

This chapter provides an overview of the research topics addressed in the dissertation and aims to briefly review some key concepts that are intrinsic to the work developed. As such, we will be touching on traffic simulation, specifically different types of simulations, a brief review of reinforcement learning, and the primary considerations to be made when applying reinforcement learning to multiple agents sharing an environment. Additionally, we briefly cover the basics of game theory to provide context regarding some parallel approaches to traffic assignment.

2.1 Traffic simulation

A simulation can be seen as a technique to study real-world systems, which are inherently complex, through digital models that mimic, to some extent, specific key characteristics of that system. As such, it is a powerful tool, particularly for processes difficult or even impossible to study from experimentation alone.

Simulations are used for a wide array of purposes and across multiple fields, such as physics, biology, engineering, transportation, finance and many others. Its value is deeply rooted in the ability to model the simplest of systems to the most complex ones, helping researchers further understand their objects of study.

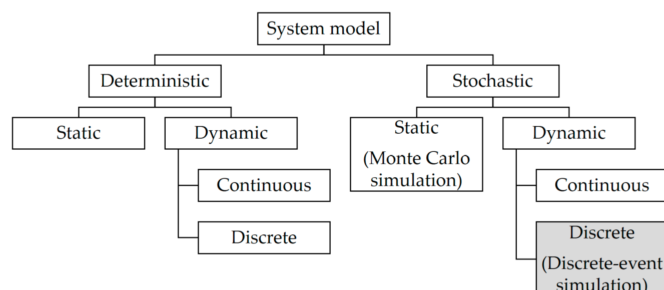

Figure 2.1: Distinction between simulation types [54]

As can be seen in Figure 2.1, there are three main dimensions with which we can classify a simulation model [29]:

- Deterministic or stochastic. A deterministic model is one where the input completely determines the output. In contrast, a stochastic model contains at least one probabilistic component.
- Static or dynamic. A static model describes a system at a single point in time. In contrast, a dynamic model represents systems as they evolve over time
- Continuous or discrete. A discrete model sees state variables change instantaneously at fixed points in time. In contrast, a continuous model sees state variables change continuously with respect to time.

In this work, the interest in simulation relies on the field of cities, namely the cities' traffic. Simulations can play a significant role in planning and optimizing a city's traffic flow by allowing inexpensive and swift iterations and prototypes on an existing model. This ability is enhanced by the multitude of problems associated with physically altering a city's road network. The system's entropy rises, traffic jams are more frequent, and it is disruptive overall.

As such, there is a wide array of traffic simulators available that model traffic in ways that can be divided into three main categories, considering the scale at which they operate: microscopic, mesoscopic and macroscopic [3] (see Figure 2.2).

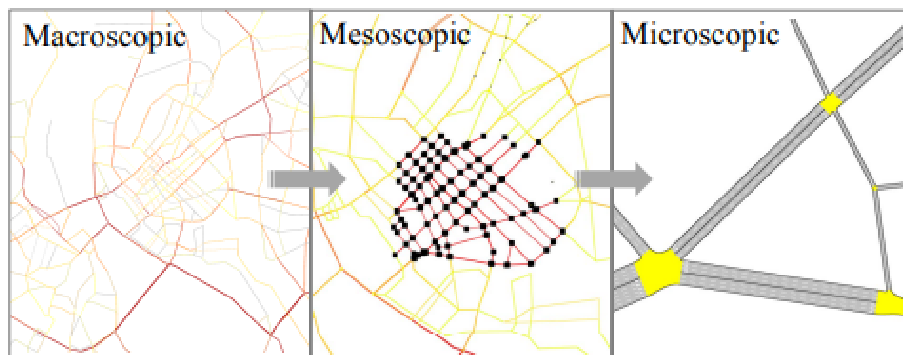

Figure 2.2: Microscopic, mesoscopic and macroscopic simulation model example[67]

Microscopic models are focused on representing in detail all dynamics and interactions of the vehicles. This implies the inclusion of many variables needed to make the closest approximation to the real world. As expected, these simulations are computationally intensive and improper for representing extensive road systems. These models describe the movement and interactions of all vehicles in the network, what includes capturing speed, acceleration, traffic signals, lane changes, etc.

Conversely, macroscopic models are designed to capture the traffic dynamics as a whole and, as such, are purposely built with scalability in mind. Vehicles lose their individuality and are

aggregated as a stream, which can be described by aggregate variables like average speed, density and flow [37].

The third category is much more ill-defined and lies between the previously mentioned models. The mesoscopic scale aims to represent an intermediate approach between micro and macro by leveraging some features of both, providing a very flexible model, as one could veer closer to either of the original models. Therefore it allows us to improve the scalability of our system by significantly reducing the level at which we simulate the environment but keeping drivers modelled individually, trading the physics-based properties of the vehicles with aggregate measurements, which allows us to study driver behaviour explicitly [8].

When characterizing traffic flow in an aggregate way, one important issue is the effect of road capacity on travel time, that is often estimated with *Volume-delay functions* (VDF)[51]. Many different types of volume-delay functions have been proposed, but one of the most widely used is the *Bureau of Public Roads* (BPR) function [43], which is defined as:

$$T_a = t_a * \left(1 + \alpha * \left(\frac{f_a}{c_a}\right)^{\beta}\right) \tag{2.1}$$

where:

T_{a} – Link travel time assigned to vehicle for link a
t_{a} – Free flow travel time for link a
f_{a} – Flow on link a per hour
c_{a} – Predicted capacity of link a in vehicles per hour

Additionally, the α and β parameters encapsulate the characteristics of link a and are calibrated from empirical data. Without such data, default values are assumed to be 0.15 and 4.0, respectively [41]. The capacity of a given link represents the number of vehicles it can contain at any given moment so that the vehicles in it can travel without experimenting relevant delays.

2.2 Game Theory

The field of game theory studies strategic interactions between rational agents, and it has permeated several areas of investigation away from its roots in economics.

A game can be defined as a scenario where players' actions impact the outcome of other players. The actions taken by a player constitute their strategy for the game. Every action has an associated payoff, which measures the player's success in a given scenario [11].

The payoff can be described as a quantifiable measurement of the player's performance in a game. As such, it is usually measured in some material reward like money or utility that a player gets from the game's result. However, utility is a subjective measure of the player's satisfaction with an outcome. Two agents might receive the same payoff, but their utility might differ.

When all players have chosen their strategy, and no player can improve their payoff by changing its strategy, we are confronted with a Nash equilibrium [40]. This property is akin to the

concept of user equilibrium in traffic theory which states that when in this state, no driver can unilaterally reduce their travel cost by switching to another route. Alternatively, we can also identify a system optimum, which provides the minimum total travel cost and consists of an optimisation problem.

The ratio between the worst possible Nash equilibrium and the system optimum represents the price of anarchy. This notion represents the degradation a system experiences due to the selfish behaviour of its agents [28].

Some properties can help better understand the differences between the many types of games. Games can be cooperative or non-cooperative depending on the existence of the ability for players to form binding contracts between them. Symmetry determines if the actions have the same payoff regardless of which player executed them. Games are said to be simultaneous if players take their actions at the same time, which means that players are not aware of the choices being made by others. In contrast, sequential games allow players to have some degree of information regarding the actions taken by previous players. If all players are completely aware of all actions taken, it is known as a perfect information game.

Games can be said to be zero-sum when the sum of the payoff for all players equals zero. For example, in a two-player game, the gain obtained by player one is equal to the loss of player two [11].

The concept of congestion games, first proposed by Robert W. Rosenthal [48], is characterized by a set of players, $\mathbb{N}$, and a set of congestible elements, $\mathbb{E}$. The strategies for each player are a subset of $\mathbb{E}$, and the payoff for each player is calculated according to the resources selected and the number of other players that selected those same resources. This game is applicable in the context of traffic if we look at roads as congestible resources and the strategy as the set of roads a player takes when going from point A to point B. By doing so, it makes sense to derive a player's payoff directly from the roads he chooses and the number of other players choosing them as well.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a popular machine learning method in which agents are goal-directed and learn from interacting with the environment. This means we do not define how the agent should solve a specific task but instead attribute rewards to actions the agent performs. The objective of the agent is to maximize the reward he receives, and as such, the agent must explore the environment and exploit actions it knows to be effective.

A basic RL agent observes the environment, picks an action leading into a new state, and then receives the respective reward for that transition. Figure2.3 is a classical representation of the loop of interactions between an agent and the environment.

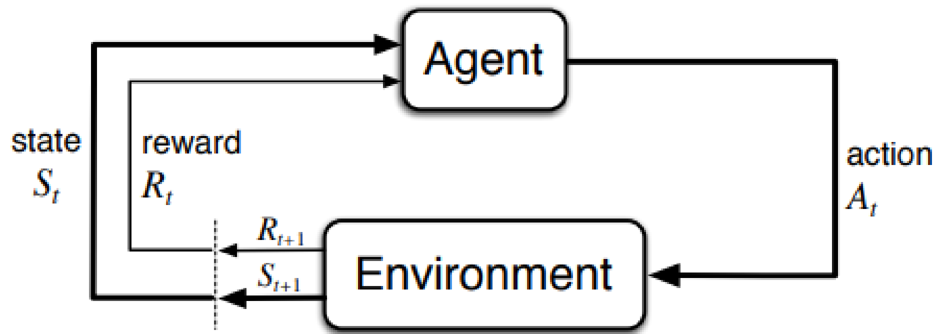

Figure 2.3: Interaction between agent and environment in reinforcement learning [59]

Apart from the environment, and according to Sutton and Barto [59], reinforcement learning systems rely on three additional elements plus an optional fourth:

- the *policy* can be seen as a mapping between perceived states and actions.
- the *reward signal* is the value that the agent receives after every action. The agent wants to maximize the total reward received.
- the *value function* encompasses the total reward an agent expects to obtain from its current state onwards.
- the *model* functions as a mirror of the environment and can be used for planning actions.

Reinforcement learning models are usually grounded on a Markov Decision Process (MDP), which captures sequential decision-making where actions have lasting consequences.

A given finite MDP can be defined by a collection of states and actions, a state transition function giving the probability that an action a will lead to state s, and a reward estimation function that predicts the reward obtained from executing action a in state s. Equations 2.2 and 2.3 show the temporal relationship between states, actions and rewards:

$$P_{ss'}^{a} = \mathbb{P}[S_{\text{t+1}} = s' | S_t = s, A_t = a] \tag{2.2}$$

$$R_{s}^{a} = \mathbb{E}[R_{\text{t+1}} | S_t = s, A_t = a] \tag{2.3}$$

where:

S – Set of states
A – Set of actions
$P_{ss'}^{a}$ – Probability that action a in step s will lead to state s'
R_{s}^{a} – Estimated reward for performing action a in state s

For a problem to comply with the conditions of an MDP, the actions must be influenced by the current state alone, and the reward attributed must be the result of applying an action to a state.

2.3.1 Multi-agent Reinforcement Learning

As a sub-field of reinforcement learning, multi-agent reinforcement learning studies the interaction between multiple agents in a shared environment.

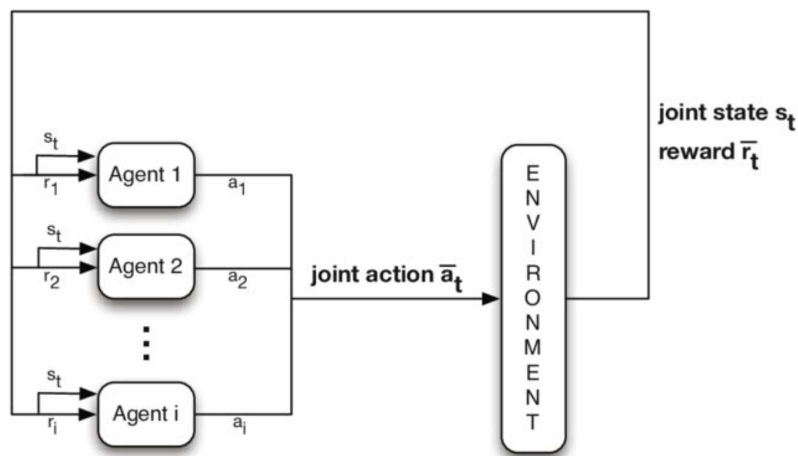

Figure 2.4: Multiple agents in the same environment [42]

Comparing Figures2.3 and 2.4 we can observe the differences between single and multi-agent reinforcement learning. In the second case, we now see that several agents take actions independently, which when combined produced the next state of the environment, which is then given to every agent, together with the specific reward for each agent, and the cycle begins anew.

We can break down the interactions between the agents into three categories: if agents have a common goal, the task is said to be *cooperative*; if agents are competing for a specific goal, the task is *competitive*; and in complex scenarios it can be the case that the task combines both cooperative and competitive features: for example, a team game has elements of cooperation between players of the same team and competition between teams.

Regarding the transition from single to multi-agent reinforcement learning, the MDP framework is replaced by an extension named Markov games [35]. It is specially designed to include multiple adaptative agents interacting with one another.

This extension is necessary because if multiple agents were to coexist in a Markov decision process environment, they would not be able to ascertain the exact impact of their actions on the result obtained since they cannot account for the actions of others. This would imply that the state transitions would cease to be Markovian and thus invalidate the results [10].

In Markov games, the probabilities and rewards obtained depend on the joint action of all agents. However, the curse of dimensionality questions the feasibility of this solution for large-scale scenarios due to the exponential growth of the state-action space.

2.3.2 Popular Reinforcement Learning Algorithms

Although multiple reinforcement algorithms are available, a few recently developed ones have gained high popularity for their simplicity and effectiveness. The Proximal Policy Optimization (PPO) class of RL algorithms [52], introduced by OpenAI, achieves results similar to or better than their state-of-the-art counterparts and appears to be much simpler to implement and tune, being now used for a wide array of tasks. The distinctive feature behind PPO is the usage of a "clip" function that penalises severe changes in the agent's policy while being optimised by maximising the expected cumulative reward. This helps make the algorithm more stable and reliable than its predecessors.

Another robust baseline algorithm for several RL problems is the Advantage Actor-Critic (A2C) developed by DeepMind [39], which aims to improve the efficiency of the original actor-critic algorithm by allowing the usage of multiple environments in parallel. This algorithm relies on an "advantage" function, which measures the difference between a given action and the average action, to rank actions by importance before updating the policy.

2.4 Summary

This chapter was designed to offer the reader a brief recollection of some critical concepts that this work heavily relies upon. As such, we covered the basics of simulations and identified the three main categories of traffic simulators; we touched on the basics of game theory and conducted a summary of reinforcement learning and the primary considerations to be taken when transitioning into a multi-agent setting.

Chapter 3

Related Work

This chapter showcases a collection of relevant research conducted in our areas of interest, namely simulation, multi-agent reinforcement learning in the context of urban mobility and some parallel usages of game theory in modelling traffic.

3.1 Multi-Agent Reinforcement Learning

3.1.1 Usage in traffic context

There have recently been numerous applications of multi-agent reinforcement learning across multiple fields of study. In this review, we will focus on approaches designed to help improve city traffic flow.

For traffic signal control, the works of Chu et al. [13], El-Tantawy et al. [17], and Wang et al. [63] offer insightful multi-agent-based approaches for large-scale scenarios. They formulate alternatives to centralized reinforcement learning solutions targeted to overcome the rapid increase in state and action space.

Striving to improve transportation efficiency and, as a result, improve traffic flow, Lin et al. [34] describe a framework for multi-agent reinforcement learning to solve a large-scale fleet management problem for online ride-hailing platforms like Uber, Lyft and Didi Chuxing. Similarly, Shou and Di [56] study the repositioning of available drivers and propose a different reward design capable of attaining a more desirable equilibrium.

The works of Li et al. [32], Xu et al. [65] and Zhou et al. [70] tackle the previous issue in a slightly different manner. Instead of focusing on the distribution of drivers across the city, they focus on the efficient assignment of orders to a set of drivers for the same types of platforms.

For vehicle routing problems, there are two main approaches in recent literature. In works from Stefanello et al. [58], Ramos et al. [47], and Zhou et al. [69], we see an agent's action as a route from origin to destination. This approach does not accurately mimic human behaviour when dealing with traffic since it relies on the premise that drivers follow a predetermined route entirely.

Alternatively, the following works model the route as a collection of decisions. This means that the action space of an agent is the list of outgoing links from the node he is currently in. This approach enables drivers to react to traffic conditions and choose their routes accordingly.

Mao and Shen [38] take a single-agent approach to this problem, which is a limitation when trying to adequately capture competitive interactions between drivers, as the agent's route choice will not have any impact on the network's congestion state.

Grunitzki et al. [21], and Bazzan and Grunitzki [4] use multi-agent reinforcement learning, but the agents learn independently. As such, agents are unaware of the presence of other agents, which would make the environment non-stationary and not Markovian [42].

Finally, Shou et al. [55] devise a scalable multi-agent solution in which agents are aware of the actions being taken by other agents. The authors take an approach similar to Yang et al. [66], where every agent interacts with an aggregate of its neighbours' actions. This approach helps circumvent the computational implications of keeping agents informed of each other in large-scale scenarios.

3.1.2 Relevant MARL frameworks

A study of available frameworks for multi-agent reinforcement learning was also conducted to help inform the decision on which platform to develop and train our agents. The platforms considered are summarised in table 3.1 according to the degree of documentation available, as well as the support for hyperparameter tuning.

Table 3.1: MARL frameworks

MARL frameworks		
Framework	Documentation	Hyperparameter tuning
MAgent2 [62]	Sparse	No
Mava [46]	Extensive	No
PettingZoo [61]	Extensive	No
PyMARL [50]	Sparse	No
RLlib [33]	Extensive	Yes

PettingZoo is a library that offers a large collection of MARL environments and a standard API to create custom ones. This library is maintained by The Farama Foundation [1], which is also responsible for MAgent2, a project geared towards MARL environments with a very large number of agents. Both these platforms are designed exclusively towards multi-agent reinforcement learning but do not offer the learning algorithms themselves.

Mava is a recent addition to this field, focusing on distributed MARL and differentiating itself by using JAX [2], a framework developed by Google to speed up machine learning tasks. This framework easily connects with environments from PettingZoo.

[1] `https://farama.org/` (Last Access: January 2023)
[2] `https://jax.readthedocs.io/en/latest/` (Last Access: January 2023)

Another framework we read about is PyMARL, developed by the WhiRL, a machine learning research group from the University of Oxford [3]. Unfortunately, this project does not appear to be under active development, and there is not a lot of documentation available.

Finally, RLlib offers a complete package regarding single and multi-agent reinforcement learning. It provides several environments, extensive documentation, and user guides for adapting environments from other platforms, such as PettingZoo or converting single-agent environments from OpenAI Gym into multi-agent scenarios.

We were mainly focused on choosing a framework that provided ample documentation since we had to create an environment from scratch to model our problem, which meant that we would likely run into multiple issues. From the more mature works with ample documentation, RLlib stood out to us, considering it has a very active community. As a bonus, it allows automatic tuning for hyper-parameters, which is an interesting option to have further down the line.

3.2 Traffic Simulation Models

Recent comparative studies of multiple existing simulators and their capabilities, e.g., the in-depth analysis of Pell et al. [45] and Saidallah et al. [49], offer a concise and detailed view of the types of configurations these simulators provide. Of the 23 simulators listed in the mentioned studies, only five fully support the mesoscopic scale. An even more important distinction between them, for academic purposes, is if they are open-source, which narrows the list down to two simulators: The first is DynaMIT [5], in which vehicles are grouped into cells that traverse links, and as such, control speed of the vehicles. The second is TRANSIMS [57], and it is characterized by using another type of model named cellular automata. In this case, the roads are discretized into cells that can be either empty or occupied by a vehicle. Vehicles follow a set of rules to determine the number of cells they cross between each step.

Outside the studies mentioned above, we also identified two additional open-source mesoscopic simulators: Mezzo [8, 7], which uses a different paradigm than the previous two mentioned, where vehicles are modelled individually and are controlled by macroscopic properties at the edges, coupled with a queue-server at the nodes to account for delays induced by traffic signals or other interactions with traffic from other directions; and Stream[14], a mesoscopic event-based simulator that processes vehicles instead of flows, calculating travel times between nodes instead of calculating every position with a fixed time step.

None of these four solutions appears to be actively developed, which, coupled with a lack of documentation, makes developing a custom simulator a more fitting option for straightforward integration with MARL.

Table 3.2 briefly summarises and lists the five mesoscopic simulators identified in the review studies mentioned, plus one that has since released support for this scale, PTV VISSIM, as well as the additional two we encountered, and finally, CityFlow [68] which despite being a microscopic

[3] http://whirl.cs.ox.ac.uk/ (Last Access: January 2023)

simulator, appears to be capable of large-scale simulations and is designed to study traffic signal control using multi-agent reinforcement learning.

Table 3.2: Simulator comparison

Simulator comparison			
Simulators	Mesoscopic	Open-source	Supports MARL
Aimsun Next [1]	Yes	No	No
CityFlow [68]	No	Yes	Yes
Cube Voyager/Avenue [6]	Yes	No	No
DynaMIT [5]	Yes	Yes	No
Mezzo [7]	Yes	Yes	No
PTV Vissim [20]	Yes	No	No
Stream [14]	Yes	Yes	No
TransModeler [15]	Yes	No	No
TRANSIMS [57]	Yes	Yes	No

Additionally, Hu et al. [23] recently presented a scalable Meso-Macro traffic simulation model that could be a useful testbed for multi-agent reinforcement learning tasks, as the simulator proposed far outperforms SUMO, approximately by a factor of 20, allowing for simulations with millions of drivers in extensive networks. The solution has not yet been made open-source, but the authors mention plans to implement an open-source "gym-fashion environment" in the future.

Finally, and moving away from specific simulator tools, we encountered a review by Wang et al. [64], which offers excellent insight into recent approaches to improve the environmental sustainability of current urban mobility plans. It includes studies of vehicle emissions models and different traffic assignment problems where ecological objectives, traffic signal control optimization and implementations of emission-oriented dynamic road pricing heavily influence route choice.

3.3 Game Theory in Traffic

Game theory has been applied to many fields in recent years. Its employment in the study of urban traffic is composed of different approaches.

The works of Ben-Elia, Klein and Levy [25, 26, 27, 31] study the emergence of cooperation in route-choosing games in a small road network. In a particular work, they show that a self-organized system can emerge from a user equilibrium state when some altruistic agents are present in the system.

The dynamics of lane changing have also been looked up from a game-theoretical perspective. The work of Guo and Harmatim [22] showcases an analysis of the lane-changing process in a congested traffic scenario. The authors describe the game proposed as a "multi-player non-zero-sum non-cooperative game". Sheikh et al. [53] devise a game-theory-based controller to estimate driver aggressiveness and identify incidents resulting from aberrant lane-changing behaviour.

Traffic signal control has also been subject to research in this field. For example, González [12] tackle the forming of queues at signalized intersections by considering a continuous Markov game. Aragon-Gómez and Clempner [2] pursue reinforcement learning for solving similar problems across multiple junctions using a similar version of the previously mentioned continuous-time Markov game.

Calderone and Sastry [9] propose a game where each agent chooses its route between nodes instead of a path from origin to destination. The game presented by the authors is of particular interest to our work since it provides a way to capture the reaction to unforeseen events.

Finally, the works by Li et al. [32], Shout et al. [55], Tanaka et al. [60], and Huang et al. [24] illustrate possible answers to the limitations of multi-agent reinforcement learning at the scope we wish to achieve. Although the works mentioned depict the route-choice interactions between multiple agents, each agent only interacts with a single agent, the environment. Based on mean-field game theory, this approach aggregates the result of the actions of all other drivers in the environment. From this approach, we gain two valuable insights:

The first is that we guarantee that the system remains Markovian as the agent is, even if indirectly, aware of the actions of all other agents. The second is that the complexity of our system decreases, and it helps achieve better scalability [32].

3.4 Gap analysis

It is our understanding that there is a need for a new open-source solution to the problem we wish to address. More particularly, we feel this work could help bridge the gap in traffic simulation, particularly regarding large-scale mesoscopic simulators compatible with multi-agent reinforcement learning experiments. This is relevant, considering it enables the analysis of expected results when making changes to a network in the real world to improve urban mobility.

3.5 Summary

From the literature review, we gathered valuable insights across the fields in which our work is inserted. We identified and analysed some simulators that operate on a similar scope to the one we intend to implement. We also identified some details we must be mindful of when considering the transition from single to multi-agent scenarios for reinforcement learning.

Chapter 4

DynamiCITY Sim: simulation-based network assignment for sustainable cities

In this chapter, we formalize the problem under study and present the research question we intend to answer. Additionally, we present a general outline of the steps taken in the proposal and detail the resources used to validate our solution.

4.1 Problem Statement

Improving urban mobility efficiency is imperative when promoting the sustainable development of modern-day cities. One of the ways we can improve urban efficiency is to introduce changes to the road network that target the reduction of commute travel time. Making these changes is not only complex but also very bothersome to commuters, which is why we find it crucial to have tools designed for conducting research in a practical and timely manner.

There is a significant number of simulators designed for this exact purpose. Still, we feel that there is a need for one that explores the unintended egoistical decisions that drivers practice daily on a city-wide scale allowing the observation of the results that changes in a network might produce. In multi-agent reinforcement learning, agents are pitted against each other, attempting to maximize the reward they receive while sharing the same environment and competing for the same resources, which could be a fitting metaphor for the interactions drivers have with each other during their daily commutes.

As such, this work joins these two areas and intends to answer the following question:

"*Does multi-agent reinforcement learning capture the decision-making processes of drivers in traffic?*"

4.2 Proposed Solution

To tackle the aforementioned research question, we propose a solution anchored on two pillars:

1. An event-driven mesoscopic simulator.
2. Application of multi-agent reinforcement learning to solve traffic assignment problems.

The decision to construct a mesoscopic simulator is based on the advantages it presents in the problem under study in relation to the use of other types of simulation models. Microscopic simulators struggle to reach the scale we intend to deliver, given the detail in which they simulate the world. Macroscopic tools represent traffic as an aggregate effect, which results in an abstraction of the notion of individual drivers. This would prevent us from studying the decision-making process of the drivers in a multi-agent reinforcement learning scenario.

To better represent the actions and decisions of our drivers, we create a scenario where the actions of different drivers are interdependent, much like in the real world. We then use multi-agent reinforcement learning to capture the decision-making processes that arise when multiple agents compete for the same resources in a shared environment.

The proposed solution, illustrated in Figure 4.2, includes two main components, named *DynamiCITY Sim* and *DynamiCITY Learn*. In a first step, the component *DynamiCITY Sim* was developed independently, responding to item 1 above. It consists of a mesoscopic simulator that allows the user to conduct simulations based on input files (see Figure 4.1).

For any given network, users may provide either a list of routes for all vehicles or an origin-destination (OD) matrix from which the simulator creates said routes. The simulation is then performed, and the results are given to the user. The entirety of this process and its implementation is thoroughly explained in Chapter 5.

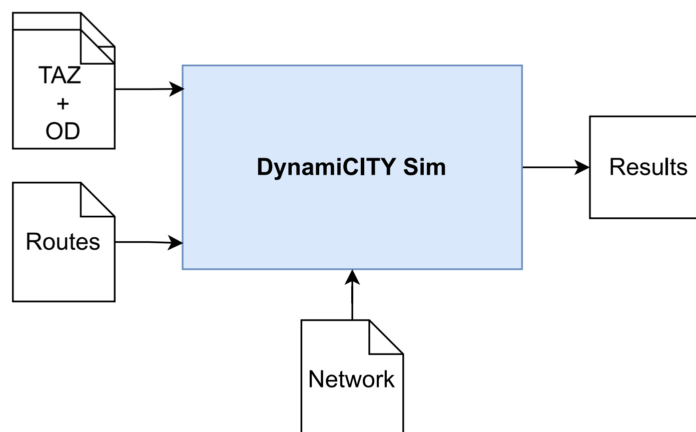

Figure 4.1: Independent simulator usage

In a second step, component *DinamiCITY Learn* was developed, and both components *DynamiCITY Sim* and *DinamiCITY Learn* were used in the proposed multi-agent reinforcement learning approach, responding to item 2 above. Chapter 6 explains the MARL process in detail. Still, from a high-level perspective, this approach allows users to apply reinforcement learning algorithms to a collection of agents to train them to complete their assigned trips as quickly as possible. This is done by having the agents plot out their paths, and once all of them have concluded their trip, a call to the simulator is made to assess the results of the actions taken. From the simulator's output, we derive the reward to attribute for each agent.

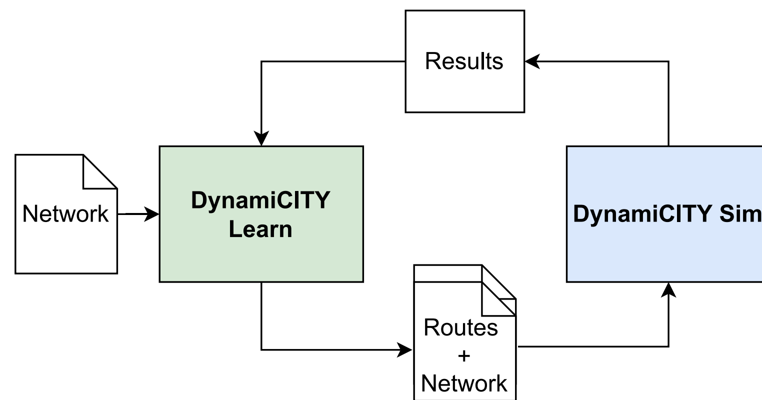

Figure 4.2: Interactions between the two modules

4.3 Validation

To validate the solution achieved by this proposal, we made separate tests for each component of the solution. The results of these experiments are discussed in detail in Chapter 7.

As a point of reference, these experiments were conducted in a machine with a Ryzen 5600X CPU, a GeForce RTX 3070 GPU and a total of 32 GB of RAM available.

4.3.1 Simulator Experiments

To verify the scalability of the simulator itself, we conducted a series of stress tests to observe how the simulator scales in time when increasing the number of vehicles being simulated, the number of edges travelled or both. For that purpose, we used a network of the city of Porto, visible in Figure 4.3, which consists of 4090 nodes and 7518 edges.

Additionally, we resorted to multiple OD matrices, each specific to a period of one hour, that were available for the city of Porto as part of the DynamiCITY project to conduct a complete 24-hour simulation of the expected traffic.

Figure 4.3: Network of the city of Porto

4.3.2 Reinforcement Learning Experiments

As for testing the multi-agent reinforcement part, several networks were used, each with specific purposes.

The first network, depicted in Figure 4.4, composed of four nodes and four edges, was explicitly designed to allow traffic between nodes *J0* and *J3* to occur in two different paths, one of which shorter than the other. The shortest path, composed of edges *E2* and *E3*, measures an arbitrary value of 200, and the longest is 300. All edges have a capacity of ten. With these values, upwards of 15 agents on the shortest path results in a travel time for the last vehicles to enter above that of the longest path.

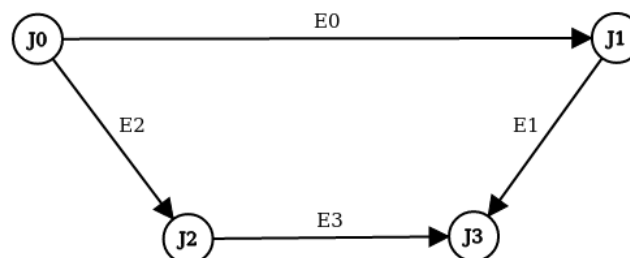

Figure 4.4: Simple validation network

For the subsequent experiments, we then turned to the Transportation Networks for Research repository[19], which contains several networks used for studying traffic assignment problems.

For each network, there is a file describing the structure and properties of the network, such as its nodes and links, a file containing an OD matrix that comprises the demand between each pair of nodes, and finally, a file with the best-known solution for the scenario. This solution consists of the flow assigned to each edge to produce the lowest total travel time. This assignment corresponds to the system optimum.

The Anaheim network, with 416 nodes and 914 edges, was mainly used to observe the behaviour the agents exhibited in a more extensive network, more specifically to understand if the agents had trouble finding a route when the network was more complex.

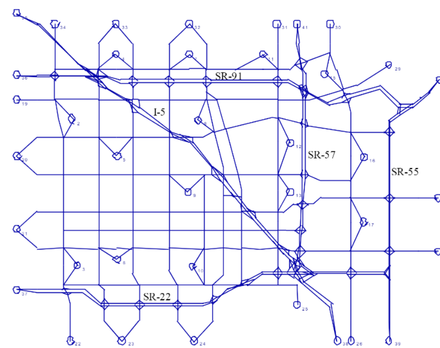

Figure 4.5: Anaheim network[19]

Finally, the Sioux Falls network is the stage for the most complete tests performed. Despite being small, with 24 nodes and 76 links, this network is widely used in traffic assignment problems despite not being considered a realistic network.

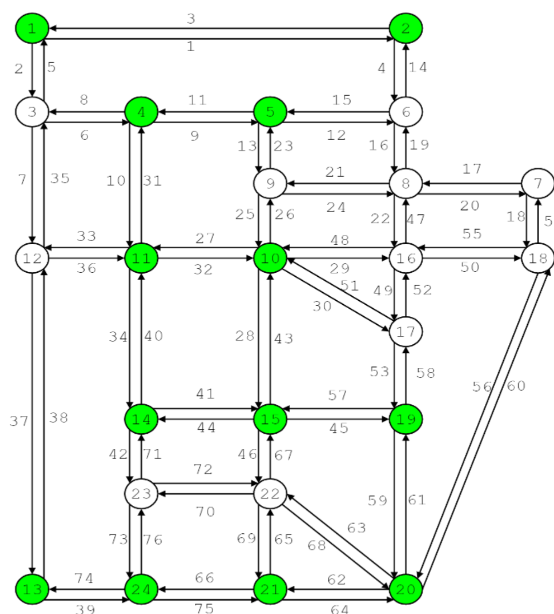

Figure 4.6: Sioux Falls network[19]

For this network, we used the accompanying OD matrix, which includes trips between every pair of nodes, which results in a total of 360,600 trips assigned. We then measure the average time spent on the network by our drivers and compare it to the optimal solution identified in previous works.

The authors of the repository explain that the OD flows correspond to a tenth of the daily flows in the original paper[30] and might be interpreted as hourly flows. This conversion has been made to allow comparison with results from papers published in the 1980s and 1990s. Additionally, units of free flow travel time correspond to 0.01 hours but are often viewed as if they were minutes[1].

4.4 Summary

To improve urban mobility, we propose the use of multi-agent reinforcement learning and a mesoscopic simulator to try and capture drivers' decision-making processes in traffic. We will test the scalability of the simulator in a network of the city of Porto and validate our MARL results in Sioux Falls, a commonly used network for traffic assignment problems.

[1] https://github.com/bstabler/TransportationNetworks/tree/master/SiouxFalls (Last Access: January 2023)

Chapter 5

DynamiCITY Sim

The present chapter describes DynamiCITY Sim, a deterministic, dynamic and discrete mesoscopic traffic simulator. As such, we cover its defining features, a brief overview of the tools used to facilitate the development phase, an in-depth explanation of the inputs and outputs of the program, the functional and non-functional requirements that outline its purpose and finally, some insights on how the simulator was implemented.

5.1 Mesoscopic Model

The model devised represents a road network in a directed graph where nodes represent intersections and edges simulate roads, as seen in Figure 5.1. Edges hold information necessary for computing the time it takes to traverse them, namely maximum allowed speed, capacity, and length.

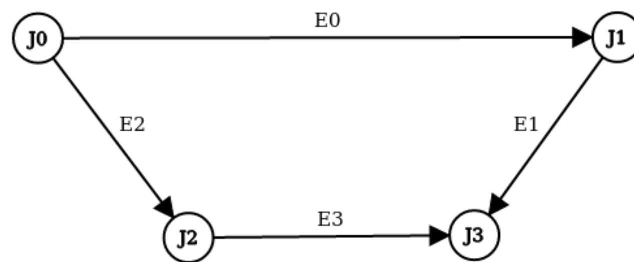

Figure 5.1: Example of a network graph

Additionally, every edge keeps track of how many vehicles have passed through it during the last hour by possessing a queue that is updated every time a driver traverses it. This allows us to estimate its flow, i.e., the number of cars that have passed through it within the last hour.

This information allows us to use the BPR function, seen previously in Equation 2.1, to estimate the time each vehicle requires to cross any given edge. This marks the distinction to a

microscopic model, allowing us to avoid having to compute every driver's position every time step.

5.2 Event Driven

Considering that our simulator is mesoscopic, we do not require a representation of the physical attributes of vehicles, i.e., position and velocity. These attributes are typically associated with microscopic simulations and a fixed timestep to update them. Simulating fixed timesteps can be a computationally demanding task, particularly when compared with an event-driven approach.

In Figure 5.2, we can see how the number of operations required for each approach differs. In our case, the only relevant instances are when a vehicle changes to another edge or terminates its route.

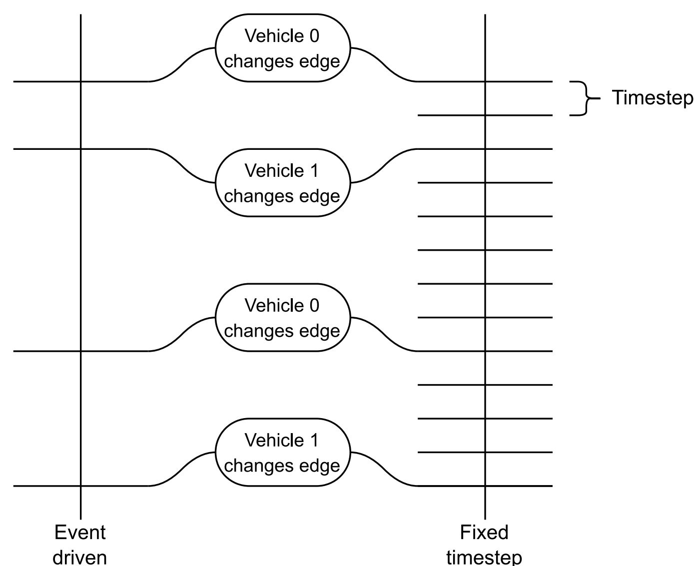

Figure 5.2: Computation instances in event-driven and fixed timestep simulations

In an event-driven simulation, we reduce the number of instances to process, which leads to a more easily scalable simulation.

Given this decision, the simulation directly corresponds to a collection of events that mark all moments where a vehicle switches edges throughout the network or terminates its route. These events are a pairing between a vehicle id and a timestamp. They are stored and maintained in a priority queue, which is ordered by these timestamps.

If two events share a timestamp, the vehicle with the lowest id always goes first. We considered incorporating a random tie-breaker process but ultimately decided against it, as it would make the simulation non-deterministic.

5.3 Software Description

The simulator developed is a command-line tool, and we have provided a help menu that can be invoked with the argument `-h` or `--help`, which produces the result visible in Figure 5.3. The usage of named arguments simplifies the process of calling the simulator as it frees the user from worrying about the order of the parameters.

The network argument, `-n` or `--network`, is mandatory and must be a path to the desired network file.

Also required is the mode argument`-m` or `--mode`, which specifies if the user is providing the routes for the drivers (`routes`) or OD matrices (`od`). In the first case, the user must also produce the routes argument, `-d` or `--drivers`, which must be a path to the file containing said routes. In the second case, the user must give the path to the Origin-Destination (OD) matrix file with `-o` or `--od` and the path to the traffic assignment zones (TAZ) file with flags `-t` or `--taz`. The contents and purpose of these files will be explained shortly.

```
Program Usage:
  --help                    produce help message
  -n [ --network ] arg      path to network file
  -m [ --mode ] arg         od or routes
  -d [ --drivers ] arg      path to drivers file
  -o [ --od ] arg           path to OD matrix
  -t [ --taz ] arg          path to taz file
```

Figure 5.3: Help menu

5.3.1 Installation

From a development standpoint, we manage the dependencies and setup the project with Conan [1], a package manager for C++, which trivializes the installation of the packages, and Premake [2], which allows us to generate project files for Visual Studio and GNU Make, thus allowing us to support both Windows and Linux for development and execution.

To use the simulator, no installation steps are required. The user simply has to run the executable with the appropriate arguments.

5.3.2 Inputs

As seen previously, the user must pass several files as arguments to the program to run a simulation. These files include the network and information necessary to obtain the routes that the drivers will be taking. We will now go into the structure of each of these files individually.

Considering that the microscopic simulator SUMO[36] was already used in the DynamiCITY project and that there was interest in also using the available network for the city of Porto, we

[1] `https://conan.io` (Last Access: January 2023)
[2] `https://premake.github.io/` (Last Access: January 2023)

decided to derive our configuration from the one used by SUMO [3]. As a result, any sumo network can be used without requiring any additional change.

The network file, exemplified in Listing 5.2, is mandatory and must follow the following XML schema:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="net" type="net"/>
  <xs:complexType name="lane">
    <xs:attribute type="xs:string" name="id" use="required"/>
    <xs:attribute type="xs:float" name="speed" use="required"/>
    <xs:attribute type="xs:float" name="length" use="required"/>
  </xs:complexType>
  <xs:complexType name="edge">
    <xs:sequence>
      <xs:element type="lane" name="lane" maxOccurs="unbounded" minOccurs=
          "1"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="id" use="required"/>
    <xs:attribute type="xs:string" name="from" use="required"/>
    <xs:attribute type="xs:string" name="to" use="required"/>
    <xs:attribute type="xs:integer" name="capacity" use="optional"/>
  </xs:complexType>
  <xs:complexType name="junction">
    <xs:attribute type="xs:string" name="id" use="required"/>
  </xs:complexType>
  <xs:complexType name="net">
    <xs:sequence>
      <xs:element type="edge" name="edge" maxOccurs="unbounded" minOccurs=
          "1"/>
      <xs:element type="junction" name="junction" maxOccurs="unbounded"
          minOccurs="2"/>
    </xs:sequence>
    <xs:attribute type="xs:float" name="version"/>
  </xs:complexType>
</xs:schema>
```

Listing 5.1: Network XML Schema

[3] https://sumo.dlr.de/docs/Networks/SUMO_Road_Networks.html#network_format (Last Access: January 2023)

Edges and nodes must be elements marked with the tag *edge* and *junction*, respectively; Edges must contain one or more elements marked with tag *lane*, which, in turn, must have the attributes *speed* measured in meters per second and representing the maximum allowed speed and *length* measured in meters and representing the total length of the road; Edges must contain attributes *id*, *from* and *to*; Edges may also carry the attribute capacity, measured by the number of vehicles per hour. If this attribute is missing, it will be estimated according to the following guidelines [16] from the Portuguese Institute of Mobility and Transportation.

$$C(l) = \begin{cases} 2200 * l & \text{if } l > 1 \\ 1700 & \text{if } l = 1 \end{cases} \quad (5.1)$$

where:

$C(l)$ – capacity assigned to the edge
l – number of lanes in the edge

Since these network files are designed for use with SUMO, several other constructs present in these network files are not helpful for us and therefore ignored when parsing the file. Most notably, all edges and nodes bearing the attribute *internal*, roundabouts and connections. Creating a network from scratch might result in something similar to the following example:

```
1  <net version="1.0">
2      <edge id="E0" from="J0" to="J1" >
3          <lane id="E0_0" speed="10.0" length="100.0" capacity="2000"/>
4      </edge>
5      <edge id="E1" from="J1" to="J2" >
6          <lane id="E1_0" speed="10.0" length="100.0" />
7      </edge>
8      <edge id="E2" from="J0" to="J2" >
9          <lane id="E2_0" speed="10.0" length="150.0" />
10         <lane id="E2_1" speed="10.0" length="150.0" />
11     </edge>
12
13     <junction id="J0"/>
14     <junction id="J1" />
15     <junction id="J2" />
16 </net>
```

Listing 5.2: Example of a network file

Our program requires the complete routes for all vehicles present in the simulation. This information can be given in two different ways. First, we can pass an XML file containing the routes for every driver. A route corresponds to a list of edges that the driver will take. Listing 5.3 showcases an example of a routes file for the network exemplified in Listing 5.2. Notice that for every

driver, the routes correspond to a string with the id of all edges to be taken, separated by white spaces. Drivers also have a timestep attribute which dictates when the route is planned to begin and is measured in seconds.

```
<drivers>
  <driver id="0" route="E0 E1" timestep="0" />
  <driver id="1" route="E0 E1" timestep="0" />
  <driver id="2" route="E2" timestep="0" />
  <driver id="3" route="E2" timestep="0" />
  <driver id="4" route="E2" timestep="10" />
</drivers>
```

Listing 5.3: Example of a routes file

Alternatively, and mainly to allow for the usage of data on the city of Porto, which was available to us in the context of the DynamiCITY project, the program can infer the routes from two additional files:

A traffic assignment zones (TAZ) file defines groups of edges that share a zone, and each of these edges must be defined as either source or sink; An OD matrix file consisting of, for each pair of zones, the expected number of trips to be made between them. If the user has provided a SUMO network, the respective TAZ file can also be used if it has been defined. Otherwise, it must be created from scratch and should be identical to the example provided in Listing 5.4

```
<additional>
  <tazs>
    <taz id="1">
      <tazSource id="1" weight="1"/>
      <tazSink id="2" weight="1"/>
    </taz>
    <taz id="2">
      <tazSource id="3" weight="1"/>
      <tazSink id="4" weight="1"/>
    </taz>
  </tazs>
</additional>
```

Listing 5.4: Example of a TAZ file

Finally, the OD matrix files are taken directly from the dataset available for the DynamiCITY project. Figure 5.4 offers as an example a snippet of one of these files.

```
$OR;D2
* From-Time  To-Time
08.00 09.00
* Factor
1.00
* Matrix begin at: 08:00  end at: 09:00
            69          1           15
            69          2           1
            69          3           0
            69          4           3
            69          5           4
            69          6           5
```

Figure 5.4: Snippet from an OD matrix file

The most relevant part of the file is the matrix itself, and the three values depicted correspond to the starting TAZ, the target TAZ and the number of vehicles inbound, respectively.

5.3.3 Outputs

The result of the simulation of the previously defined example can be displayed in two different manners. The first corresponds to a detailed outlook of all drivers and routes. It shows, as can be seen in Figure 5.5, for each driver, how much time it spends in every step of the route and the flow measured at that edge before the driver entered it.

```
DRIVER: 0 | Total time: 20
        EDGE ID: E0 | TIME: 10 | FLOW: 0
        EDGE ID: E1 | TIME: 10 | FLOW: 0
DRIVER: 1 | Total time: 20
        EDGE ID: E0 | TIME: 10 | FLOW: 1
        EDGE ID: E1 | TIME: 10 | FLOW: 1
DRIVER: 2 | Total time: 15
        EDGE ID: E2 | TIME: 15 | FLOW: 0
DRIVER: 3 | Total time: 15
        EDGE ID: E2 | TIME: 15 | FLOW: 1
DRIVER: 4 | Total time: 15
        EDGE ID: E2 | TIME: 15 | FLOW: 2
```

Figure 5.5: Detailed output

```
0:20
1:20
2:20
3:15
4:15
```

Figure 5.6: Simplified output

Specifically for usage in conjunction with the multi-agent reinforcement learning system developed, a simplified version was also created, visible in Figure 5.6, which lists each driver's id and their cumulative travel time.

5.4 Requirements

Before implementation began, the requirements listed below were identified:

- Compatibility with preexisting networks available in the project
- Compatible with the MARL component of this work
- Highly scalable
- Simulation based on a given OD matrix
- Simulation based on a given list of routes per driver

5.5 Implementation

The simulator was implemented in C++ and can be broken down into three classes which will be further explained below. Besides the standard template library, we also use some libraries from Boost[4] and Pugixml[5]. Pugixml is used for assisting in the parsing of the XML input files, while Boost holds a more integral role in our program, as is explained in the following subsections.

The program's entry point resides obviously in the main function and is the first place we use Boost. To simplify parsing the arguments given by the user, we make use of the *Boost Program options* library[6].

5.5.1 Driver

The driver class represents a vehicle operating in the simulation and, as such, holds information regarding the route to execute. During the simulation, the driver collects information regarding the path taken by holding a map with the time spent on each edge, which is used while assembling the final output.

Each driver is responsible for providing its output of the simulation, either in the detailed format shown in Figure 5.5 or in the condensed version, like in Figure 5.6 for usage with the MARL module.

5.5.2 Network Model

This class is responsible for constructing an intermediate representation of the information contained in the input files. This can be broken down into two main steps. The first relates to the parsing of the network file and the respective creation of the network. The second pertains to parsing the routes the vehicles are to execute, either by reading a routes file directly or calculating the routes to fulfil a given OD matrix.

[4] https://www.boost.org/ (Last Access: January 2023)
[5] https://pugixml.org/ (Last Access: January 2023)
[6] https://www.boost.org/doc/libs/1_81_0/doc/html/program_options.html (Last Access: January 2023)

Graph Creation

Creating the graph involves parsing the network file and creating the appropriate data structures to represent this information. We use Boost's adjacency list implementation[7] to represent the graph itself. This implementation allows `structs` to represent properties of edges and nodes, which we take advantage of. The adjacency list is filled as we parse the nodes and edges that make up the network we wish to represent.

The vertex property holds an id and an integer to enable, in the future, the implementation of delays that might represent traffic lights or other types of interactions at play in intersections. Regarding the characteristics of the edge itself, we store the max allowed speed, the known capacity of the edge, its length and the VDFs to be used when estimating the travel time.

Edges contain a queue representing the flow of the edge, which is only updated when required. To update them, every time a vehicle enters an edge, we insert a timestamp into the queue and remove all timestamps older than one hour. To give an example, in Figure 5.7, when a driver enters an edge with a timestamp of 4000, we remove any timestamp more than one hour old, i.e., 4000 minus 3600 which is 400, therefore removing the first two timestamps, one and ten.

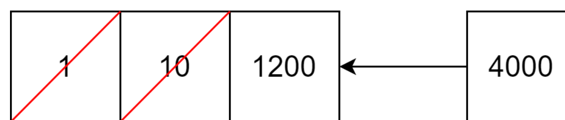

Figure 5.7: Example of an edge updating its flow

Routes

Responsible for instantiating the drivers with their respective routes, this process is done differently considering the usage mode the user has selected.

Suppose the user has selected the option of providing routes. In that case, it is simply a matter of parsing the provided XML file, reading the routes for each driver from it and instantiating the drivers with this information.

If the program runs on OD matrix mode, some steps are taken to ensure that routes are created from the provided files. First, we must parse the TAZ file and its information in the appropriate data structures, making the distinction between source and sink edges for each zone defined.

We then proceed to parse the OD matrix provided and, for each demand, instantiate the specified number of drivers. Since the drivers need to know the route they will take, we create this route by running Dijkstra's shortest path algorithm. This is done through the boost library, as the graph structure we previously created already offers this function.

Special care is taken when selecting origin and destination nodes. We take the origin node from one of the source edges and the destination node from one of the sink edges.

[7] https://www.boost.org/doc/libs/1_65_1/libs/graph/doc/adjacency_list.html (Last Access: January 2023)

5.5.3 Simulator

Finally, the simulator class encapsulates the heart of the program and is characterized by the main loop of the simulation. Essentially, we continuously process events at the top of the priority queue.

This process is done by reading the event at the top of the queue, making updates to the route of the driver associated with the event and triggering the mechanism that keeps the edge flow updated.

Finally, the event is reused by using the BPR function and the current state of the edge that will be traversed to calculate the time to travel it. This value is added to the original timestamp, and the event is reinserted into the queue, which naturally keeps itself ordered.

If the driver has reached its destination, it is not reinserted into the priority queue, as expected, and the loop ends when all events have been processed.

5.6 Summary

This chapter contains a description of the tool developed and its proper usage. We define the input files necessary for a simulation and the available output types. Finally, we list the requirements that drove the development process and offer implementation details regarding the main entities.

Chapter 6

DynamiCITY Learn

This chapter serves to relay the complete process of transposing our problem into an environment suitable for reinforcement learning algorithms. To do so, we portray the incremental steps of going from single to multi-agent reinforcement learning and the challenges encountered in the process.

Having selected RLlib as the library to work with, we chose to model our problem in an Open AI Gym environment since RLlib completely supports these types of environments and allows their usage straight out of the box.

6.1 Single-agent Environment

Before working with multiple agents, we created an environment for a single agent. This enabled us to test the available algorithms and confirm that the way we designed it was suitable for an agent to find the optimal path from origin to destination. The creation of this environment can be divided into two parts. First, we defined the environment itself, and then we implemented the required functions to manipulate the environment.

6.1.1 Defining the environment

The first step to creating an environment is to define in what shape we model the observations of the world and in what format actions are depicted.

The `observation space` attribute dictates the structure and the possible values that can be observed in any state of the environment. We defined this attribute as a two-dimensional binary matrix resulting from concatenating the network's adjacency matrix with two additional rows. The first depicts the node where the agent currently is, which translates to a row of zeroes except the element index corresponding to that node, and the latter is the same but for the node that the agent is trying to reach.

The `action space` attribute, responsible for dictating what constitutes an action, was defined by calculating the highest outward degree present in the network, or rather, determining how many outward edges the node with the most outward edges has. The possible values for actions are then the first n positive integers starting at zero, where n is the previously calculated value, and

thus, they correspond to the index of the edge to select. In cases where a node has fewer edges than the value calculated, we take the modulus of the action by the number of outgoing edges in the present node, making it so that every action leads to the selection of a valid edge.

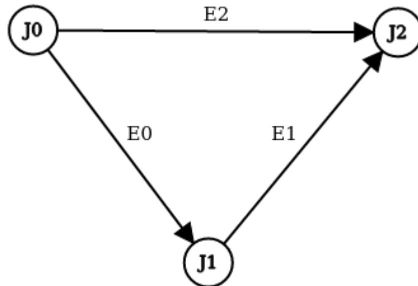

Figure 6.1: Example network

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 6.2: Possible state observation

Taking Figure 6.1 as an example, the observation of an agent at node *J0* who wishes to go to node *J2* corresponds to the matrix in Figure 6.2. Notice that the first three rows correspond to the graph's adjacency matrix; the fourth marks the position the agent currently holds, and the fifth is the desired position for the agent. In this case, there are only two possible values for actions, zero and one, since the most connected node is *J0*, which has two outward edges. If an agent were to be in node *J1*, he would have two different actions available but they would both point towards edge *E1*.

6.1.2 Interacting with the environment

This final step consists of three functions that must be implemented: `init`, `reset` and `step`. The first one, `init`, is responsible for creating the environment, accepting all additional parameters needed. In our case, the parameters of relevance include the origin and destination nodes, a previously computed array of weights for every pair of nodes, whose purpose will be explained further below, and a value limiting the number of steps that can be taken in any episode.

The second function, `reset`, ensures that the environment is returned to its original state every time a new episode occurs.

Finally, the `step` function holds most of the logic of the environment. It receives as an argument an action and is responsible for computing the resulting state of the environment and returning the associated reward. First, we need to translate the action, an integer value, into an edge to add it to the agent's route. The format in that we defined actions raised some problems, and additional logic had to be implemented to deal with illegal actions.

Some nodes are not as connected as others, resulting in cases where an agent in a node with, for example, two outward edges is being instructed to take the third edge. These cases are solved by taking the modulus of the action by the number of outgoing edges in the present node. In the example below, the third edge would be interpreted as being the first one.

After ensuring that all actions are correctly interpreted, we then analyse the results of said action by checking some conditions, namely if the agent has reached a dead end with no way to reverse or if he has taken an edge previously traversed. The episode is terminated early in both cases, and a zero reward is attributed. We then ascertain if the agent has reached its destination. If it has, the agent writes its route into the appropriate file.

Finally, we execute the simulation by initiating a child process, awaiting its termination and, from its output, determining the reward to be given, which is calculated as follows:

$$R_a(s,s') = \frac{T_{ff}}{T} \tag{6.1}$$

where:

$R_{\text{a}}(s,s')$ – Reward from transitioning from state s to state s' with action a
T_{ff} – Freeflow travel time for the shortest path between start and finish
T – Total travel time registered in the simulator

If the agent has not yet reached its destination, a different reward is given, designed to encourage the agent in the right direction but crafted to be impervious to exploitation. For every agent step, we measure the shortest path distance between the node where the agent was before and after taking its action.

$$R_a(s,s') = \frac{(D_s - D_{s'})}{D} \tag{6.2}$$

where:

$R_{\text{a}}(s,s')$ – Reward from transitioning from state s to state s' with action a
D_{s} – Shortest path distance to the destination for state s
$D_{\text{s'}}$ – Shortest path distance to the destination for state s'
D – Shortest path distance between start and finish

This enables us to award an agent that has found a path between start and finish a cumulative reward of one regardless of the path chosen. If the agent did not manage to conclude his journey, the reward amounts to how close he got to his goal.

6.2 Multi-agent Environment

Following RLlibs's guidelines for creating a multi-agent scenario, we have created a subclass of the library's `MultiAgentEnv` class. This process involves creating new `init`, `reset` and `step` functions. In practice, we instantiate multiple agents belonging to the first environment and then call their respective functions accordingly.

This transition raised issues that had to be resolved, resulting in changes to the original environment. First, upon some experimentation, we verified that a simpler version of the `observation space` representation produced better results and saved some memory. The final solution was to

remove the adjacency matrix, leaving only the previously explained two rows. We also added the id of the agent in question.

Additionally, the interaction with the simulator had to be remade for multiple reasons. First, agents could no longer be responsible for launching the simulation, as all agents have to participate in the same simulation. Second, agents should not be responsible for writing their route on the configuration file, as this would mean multiple accesses and worse performance.

The last change was minimal and resulted in simply making these operations when all agents had completed their last step before the call to the simulator. The first one, however, was much trickier and had multiple iterations before we reached a solution that depicted what we wanted to happen and went as follows:

The first approach was to await the termination signal from all agents and then issue the command to simulate and hand out the resulting awards. This solution was invalid since no rewards could be given to agents that had already declared they were finished. This meant it would only work properly when all agents terminated their routes in the same step.

The next attempt tried to answer directly to the problem encountered. If rewards cannot be given after termination, then all agents that have concluded their route now withhold the termination signal until all agents have completed the route. This solution was viable, but agents were not directly rewarded for completing their route. Instead, agents received a reward after spending some steps not acting, sometimes too far removed from the step where it happened. Some preliminary testing here revealed poor results that may stem from the fact that the reward could not be easily traced to a particular action.

These results led us to believe that we could not solve this problem directly during the `step` phase, which led us to a native feature of RLlib, callbacks, for a possible solution. According to the documentation read, callbacks can be called at various stages of the training process. The two phases that caught our attention were the `on episode step` and `on episode end`.

These callbacks enabled us to do something not yet considered. Instead of granting an additional reward, we could now replace previous rewards. This was the missing piece to ensure rewards were given in the appropriate step. The most natural phase to call the simulator was at the end of the episode, but we quickly realised that either changing or adding rewards at this stage was already too late for them to be considered learning-wise. So the final solution was to check every step to see if all agents had concluded their route. If so, we launch the simulator subprocess[1] and attribute the resulting rewards to all agents. As a minor consequence of doing this process, we need to force agents to do an additional step to allow this callback to trigger. This means that the signal sent when all agents have finished their route is, in fact, sent one step after it happens, which terminates the episode. To ensure that we replace the accurate reward, agents track which step they have reached their destination.

Figure 6.3 depicts a very simplified version of the control flow our agents experience during the training process, highlighting the instance in which the call to the simulator is made and the roles of the main functions we discussed previously. Finally, the blue box corresponds to the call

[1] `https://docs.python.org/3/library/subprocess.html` (Last Access: January 2023)

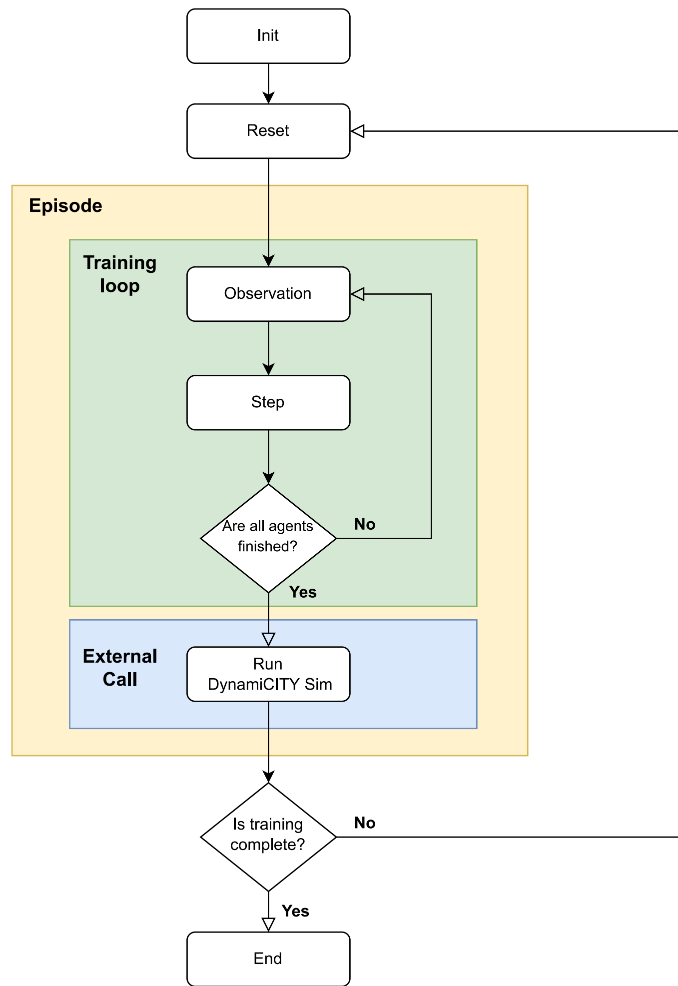

Figure 6.3: Training flow chart

to the simulator that is made during the callback that triggers during the artificial extra step that we mentioned previously.

While testing the environment with large networks and their OD matrices, we quickly realized that we could not field the number of agents required and had to devise a solution. For example, for the Sioux Falls network, which has a total of 360600 trips planned, we made it so that each of our agents is responsible for one hundred vehicles. This means that after concluding their route, each agent injects into the routes file one hundred copies of the route chosen. Their reward is calculated from the average time those one hundred vehicles completed their routes.

6.3 Training Process

The training process is conducted in a Jupyter Notebook, allowing fast experimentation and simple visualization of results. The process is very straightforward, thanks to the flexibility provided by RLlib's Python API. We have identified three stages for this process: a setup phase, the training phase and an additional step for data collection.

6.3.1 Setup

First, we need to read the network file, define the respective environment and create a configuration dictionary for the chosen algorithm. We mostly used PPO, which can be configured as depicted in Listing 6.1. This example does not showcase all available variables that can be configured, but it encapsulates the ones perceived as most relevant at the time of writing.

```
config = {
    # Environment (RLlib understands openAI gym registered strings).
    "env": "MultiAgent",
    'callbacks': SimulationCallback,
    # Use 2 environment workers (aka "rollout workers") that parallelly
    # collect samples from their environment clone(s).
    "num_workers": 1,
    "num_envs_per_worker":1,
    "framework": "tf",
    # Tweak the default model provided automatically by RLlib,
    # given the environment's observation and action spaces.
    "model": {
        "fcnet_hiddens": [256, 256],
        "fcnet_activation": "relu",
    },
    "batch_mode": "complete_episodes",
    "train_batch_size": 2 * horizon * num_agents,
    "rollout_fragment_length": 2 * horizon * num_agents,
    "sgd_minibatch_size": 2 * horizon * num_agents,
    "horizon": horizon,
    "lr": 0.0005,
    "gamma": 0.95,
    "lambda":0.90,
}
```

Listing 6.1: Example of a configuration dictionary for PPO

Regarding the values presented, the most relevant to explain here is the usage of complete episodes for *batch mode*. This is the only argument that must not be altered, as episodes must not

be truncated during an episode. This is because the core of our reward system comes from the simulation, which only happens at the last step. If an episode were to be evaluated without this step, the result would be unpredictable.

We advise the reader to consult the official documentation[2] for a complete explanation of the remaining parameters.

6.3.2 Training

After selecting and configuring the algorithm to be used, commencing and observing the training process results is quite simple. As shown in Listing 6.2, all it takes is to initialize the appropriate `Trainer` and call its `train` method. We select a few preliminary metrics to print after each iteration to provide a quick overview of the results. We also save the model created, in this case, once every ten iterations, to allow for its usage later.

```
s = "{:3d} reward {:6.2f}/{:6.2f}/{:6.2f} len {:6.2f}"
agent = PPO.PPOTrainer(config=config, env=select_env)
for n in range(N_ITER):
    result = agent.train()
    print(s.format(
        n + 1,
        result["episode_reward_min"],
        result["episode_reward_mean"],
        result["episode_reward_max"],
        result["episode_len_mean"],
        ))
    if n % 10 == 0: #save every 10th training iteration
        checkpoint_path = agent.save()
```

Listing 6.2: Creating and running a RLlib Trainer

The output throughout the training process is showcased in Image 6.4 and pertains only to cumulative values of rewards that our agents achieved during this process. For further analysis, RLlib stores the training results and the hyper-parameters used in a folder that we can specify.

```
1 reward -349.42/-312.62/-275.82 len  10.00
2 reward -349.42/ 23.57/405.66 len  10.00
3 reward -349.42/450.11/1358.84 len  10.00
4 reward -349.42/1022.66/2214.42 len  10.00
```

Figure 6.4: Output throughout training

[2] https://docs.ray.io/en/latest/rllib/rllib-algorithms.html (Last Access: January 2023)

Listing 6.3 demonstrates how to load a previously saved model. First, we need to recreate the `Trainer` that was used and then call its `restore` method, passing as an argument the path to where the model was saved.

From there, we can then ask the model to compute all necessary actions to assign all trips to create the routes that serve as input to the simulator.

```
env = MultiAgent(graph)
action = {}
obs = env.reset()
done = False
episode_reward = 0
while not done:
    for agent_id, agent_obs in obs.items():
        action[agent_id] = agent.compute_action(agent_obs)
        obs, reward, done, info = env.step(action)
        done = done['__all__']
        episode_reward += sum(reward.values())
```

Listing 6.3: Loading a model and performing one episode

6.3.3 Data Collection and Visualization

To allow for a better understanding of what our agents are doing throughout the experiment, we collect the state of the network every time a simulation happens. From this information, we create a graphical representation of the state of the network using functions from Networkx, Shapely and Matplotlib python packages, colouring each edge according to the ratio between the traffic flow and capacity as seen in Figure 6.5.

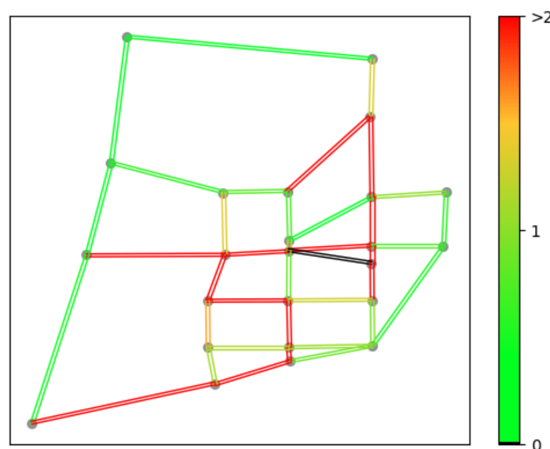

Figure 6.5: Snapshot of Sioux-Falls network with Dijkstra's shortest path assignment

Edges that have not been used are coloured black. Edges with flow lower than their capacity are coloured green, showing that drivers are not experiencing any delays in the network, while edges with higher flow get progressively redder. Keep in mind that the BPR function is not linear, which means that having one edge severely saturated has a drastically higher impact than multiple edges slightly over capacity.

In these figures, edges drawn in black represent roads never taken, and a gradient between red, yellow and green is used to demonstrate the level of saturation for each link.

Additionally, we use the last episode to gather all metrics required to make a direct comparison not only with the results available in the Transportation Networks repository but also with a hypothetical scenario of shortest path assignment to all drivers. This information is then written to a spreadsheet to facilitate further analysis.

6.4 Software Description

This module consists of a series of Python Jupyter Notebooks that hold all the necessary steps to load a network, create the environment and train multiple agents to take the most efficient route possible.

6.4.1 Installation

To reproduce the experiments, the user should use the provided requirements file to install all necessary python packages. Although it is not required, it is also possible to train with a graphics processing unit (GPU), but it requires some additional steps. We advise the user to follow the official guidelines from Tensorflow on how to use a GPU for training[3]. Once the user confirms that Tensorflow can successfully access the GPU, it should also be possible to use it with RLlib.

Although the usage of this hardware is usually linked with a significant increase in performance, in our case, preliminary tests showed that the usage of the GPU led to slower training, and for that reason, we did not make use of it.

6.4.2 Dependencies

Selecting RLlib as the library to work with came with the strong suggestion to use Gym from open AI, as it was the most direct and documented way to create an environment for our problem. RLlib supports both Tensorflow and PyTorch internally, but we decided to use Tensorflow since we had previous experience with it. Finally, we strongly use Networkx, a Python package for creating, manipulating and studying networks.

[3]`https://www.tensorflow.org/guide/gpu` (Last Access: January 2023)

6.5 Summary

This chapter entailed the process of creating a custom OpenAI Gym environment for our problem and its adaptation to support multiple agents within the RLlib framework. Additionally, the interactions between the training framework and our simulator are explained.

Chapter 7

Results and Discussion

In this chapter, we describe the experiments made to validate and test the simulator and also its usage in the MARL scenario, and the respective results. Additionally, we offer some empirical insights regarding the hyperparameters that led to better results for our machine learning processes

7.1 Simulator Experiments

The first tests were made on the simulator itself to assess the proportions at which the simulator scales. Stress tests were made in cases where the number of vehicles being simulated, the number of edges travelled or both increased. To accomplish this, we used the network of the city of Porto presented in Figure 4.3 and generated random routes for every driver. The time to assemble the routes and perform the simulation was measured separately.

To measure the impact that the number of drivers has on the execution time of the simulator, we conducted several mock simulations where all vehicles had a random route with the same number of steps, in this case, ten. In every simulation, we increased the number of drivers by a complete order of magnitude, resulting in data collected from ten drivers up to ten million drivers.

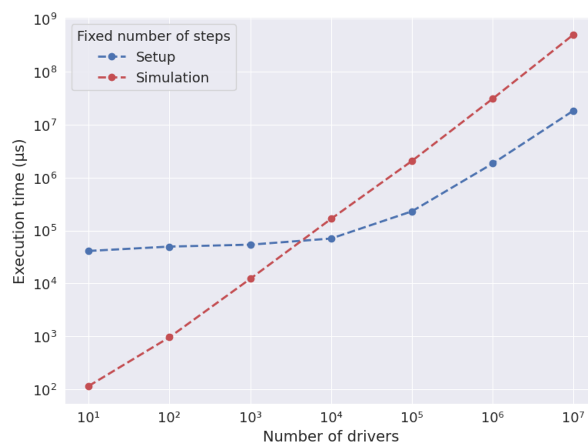

Figure 7.1: Execution time of routes with 10 steps

As for the effect of the number of steps in the route, we conducted an identical experiment, this time maintaining a fixed number of drivers, in this case, ten, which executed random routes that varied by the same factor as we did for the number of drivers in the previous point.

Although we estimate that the size of the shortest path between the two farthest points in the network resides in the neighbourhood of 150, we decided to simulate routes of up to ten million steps to observe the scalability of the simulator thoroughly.

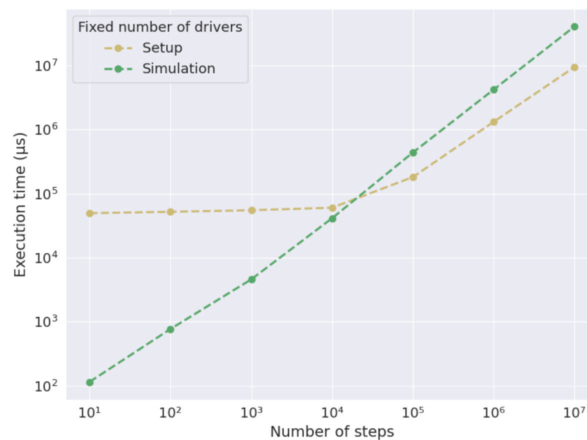

Figure 7.2: Execution time of 10 drivers

The results observed in Figures 7.1 and 7.2, in logarithmic scale, match our original prediction for the simulator's performance, especially regarding the execution time of the simulation phase. Considering that the simulator's core can be boiled down to a loop that processes events, we expected the impact of the number of drivers and steps to be similar and close to linear. For example, one driver executing a route of ten steps translates into ten events, which should be approximately identical to ten drivers performing one step.

Our limiting factor appeared to be the amount of memory required to simulate these large scale simulations. To confirm this suspicion, we measured the amount of memory being used during the execution to observe the space complexity of the solution devised. To achieve this, we read the `PROCESS_MEMORY_COUNTERS` structure[1], more concretely the `PeakWorkingSetSize` field, which keeps track of the maximum value registered during execution. This peak value was registered after the setup phase and during the simulation itself.

Once again, we expected this value to grow linearly with the number of vehicles and the length of their routes, which was verified, as seen in Figure 7.3. Increasing the number of drivers is much more taxing as it involves the creation of additional objects, as opposed to increasing the size of the structure that holds the routes for each agent.

[1] `https://learn.microsoft.com/en-us/windows/win32/api/psapi/ns-psapi-process_memory_counters` (Last Access: January 2023)

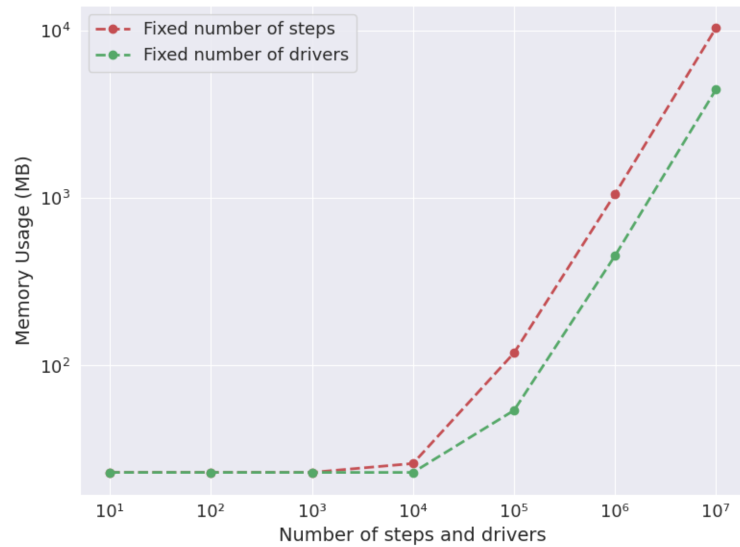

Figure 7.3: Memory consumption for both scenarios

Taking into consideration all three graphs, an interesting correlation can be found. At around the thousand mark for both quantities of steps and drivers, we can observe a change in behaviour for the setup and memory measurements. These values coincide with memory allocated measured at above 30 MB, the amount of L3 cache available in the processor used.

Additionally, we made simulations while studying the effect of increasing both the number of vehicles and steps in conjunction, which can be seen in the heat map depicted in Figure 7.4, which further helps perceive the similarity between these two variables in terms of their effect in the performance of the simulator.

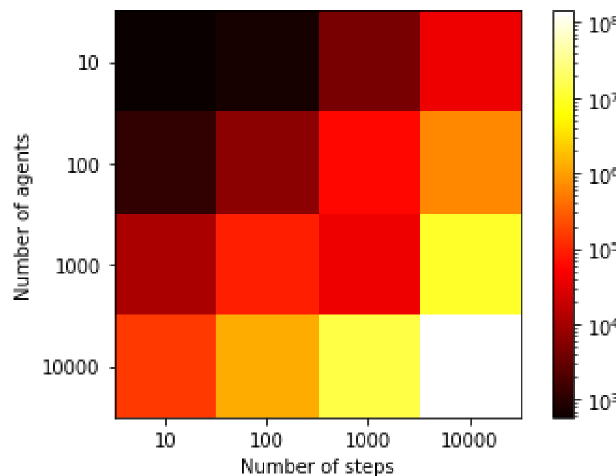

Figure 7.4: Simulation time when increasing the number of vehicles and steps

Finally, we conducted a complete 24-hour simulation for Porto city using the network and OD matrices provided by the DynamiCITY project. For the demand values provided, we ran two

simulations, one with the strict value and a second with ten times the demand for each trip.

Table 7.1: 24-hour simulation for the city of Porto

Drivers	Setup(s)	Simulation(s)	Total(s)	Memory(MB)
135229	9.25	9.12	18.37	1007
1352290	10.80	162.07	172.88	9802

To analyse the simulation results, we can draw the network at different hours during the simulation. For example, Figures 7.5 and 7.6 showcase the differences in the network between eight and nine in the morning, hours when traffic is abundant. We should mention that to achieve the result presented, we have applied a factor of 6 in the number of trips generated between each OD pair. This value was encountered manually to get a result that matched our idea of the traffic felt in the city during these hours.

Figure 7.5: Traffic prediction for Porto at 8 am

The need for this factor could stem from multiple sources, namely the fact that we do not have empirical data to assess the condition of the roads properly and, therefore, their capacity, which appears to have been overestimated by us. Additionally, in the files we received, several journeys cannot be completed, as there is no possible path between some pairs of nodes.

Although this simulation is not realistic, as it assumes that all drivers follow the shortest path to their destination, which results in a large number of roads left unused, the general essence of the traffic felt in the city during these hours appears to be captured in this representation.

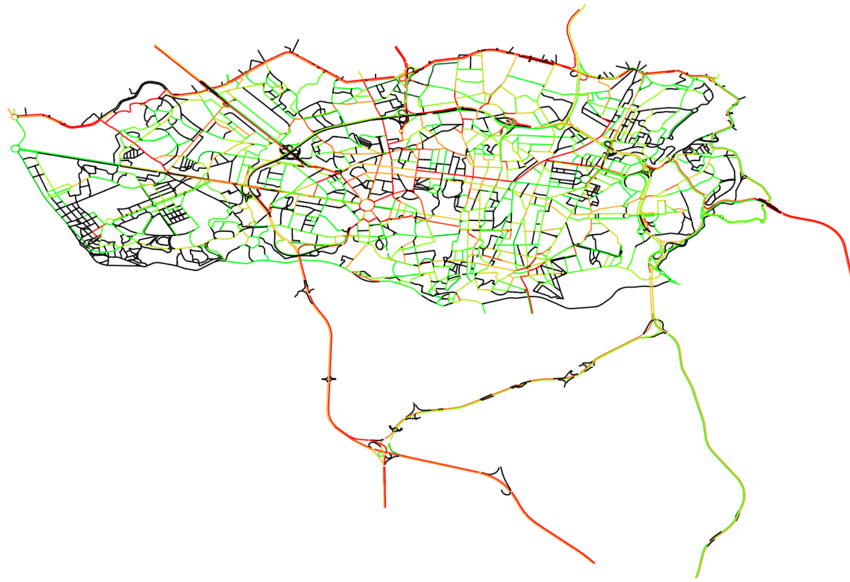

Figure 7.6: Traffic prediction for Porto at 9 am

Looking closely at the second figure, it is possible to observe the difference of traffic in roads leading into and out of the city. Looking at the Arrábida bridge, for example, we can see more congestion in the side leading north, into the city, than the side going south towards Vila Nova de Gaia.

7.2 Reinforcement Learning Experiments

To assess the capabilities of the MARL models trained, we conducted several experiments, each with different goals.

Scenario with a small network

The first and arguably most crucial test to validate the model was conducted in a very small network with a limited number of agents sharing both the starting and terminating nodes.

In this network, shown previously in Figure 4.4, agents can take only two paths, one shorter than the other. The purpose of this experiment is twofold: to ensure that the agents consistently converged on the shortest path; to observe what would happen if the most straightforward path were to become so saturated that the longest way became the fastest of the two.

To achieve this, we ran the training process multiple times, increasing the number of agents present in the network to try and catch the moment when some agents started to choose the longest path and observe if, as the network got more saturated, the solution that our agents converged to was the one we expected. This experiment was conducted with PPO and A2C to determine the most promising candidate for future tests.

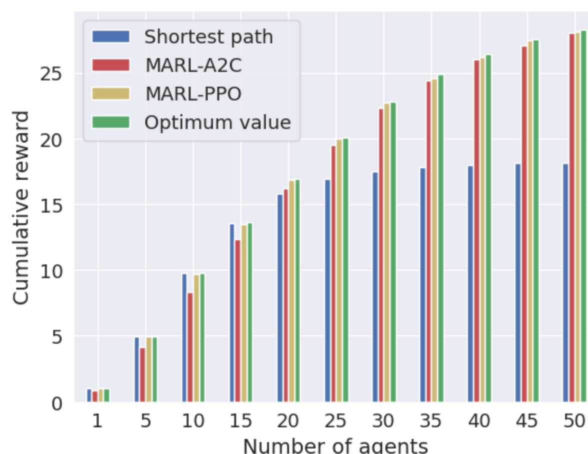

Figure 7.7: Cumulative reward obtained in saturation test with PPO and A2C

As seen above, in Figure 7.7, up to 15 drivers, the solution that the agents come up with falls a bit behind the optimal result. However, when the network becomes saturated, the agents vastly outperform a simple shortest-path solution as they begin to split between the two paths.

Comparing our agents with the optimum value we manually calculated, we can see that the agents come very close to this level, especially when more agents are present in the network.

This was the key indicator that our system was ready to advance to the next stage. Although the differences are minor, it was clear that the agents were trying to maximize their reward and behave close to what was expected. When lanes started getting congested and the longest path became more attractive, some agents chose the longest route.

For the subsequent experiments, we then turned to the Transportation Networks for Research repository [19], which contains several networks used for studying the traffic assignment problem. For each network, there is a file describing the structure and properties of the network and its nodes and links, a file containing an OD matrix that comprises the demand between each pair of nodes, and finally, a file with the best-known solution for the scenario.

Stress tests scenarios

First, we decided to conduct a stress test to ascertain the relationship between network size and the number of agents we could field and evaluate the route that the agents pick when confronted with more extensive networks. We compared the outcome in the Sioux Falls and Anaheim networks from the repository and the Porto network to perform this test.

The number of agents we can run in these scenarios is not enough to provoke saturation in any edges, which means that the shortest path will always be the most efficient path the agents can take. Knowing this, we measured the quality of the path created by our agents as the proportion between the travel time experienced and the travel time of the shortest route.

Considering that we are now dealing with larger networks, we saw ourselves strongly encouraged to limit the number of steps that agents could take in every episode. This meant that agents

might not be able to complete their path, which is why we also measured the success rate for finding a path as the proportion of agents that managed to establish a path in the number of steps granted.

First of all, for the Sioux Falls network, in every scenario, all agents could find a path, whereas, in the Anaheim network, we registered an average of 0.5% failures to reach the destination. As for the Porto network, we could not complete the experiment, and the results were vastly inferior. This is due to this network's high complexity and size, which proved too difficult for our agents and our hardware setup. The success rate registered for all experiments can be seen in Figure 7.8.

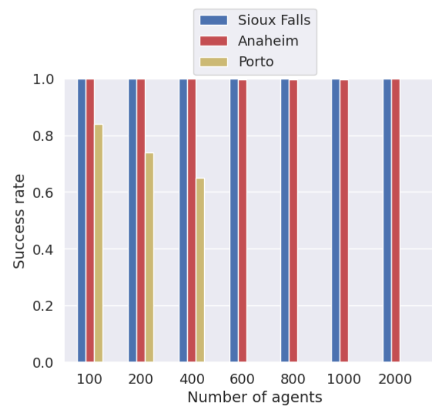

Figure 7.8: Path discovery success rate for Sioux Falls, Anaheim and Porto networks

The results of the path evaluation are visible in Figure 7.9 and refer only to the agents that managed to complete their routes. For the Sioux-Falls network, our agents achieved, on average, a score of 93%, while for the Anaheim network, the result is a bit lower, standing at 87%. Finally, for Porto, results were only collected for the first three cases, which revealed that not also did we see a low amount of agents being capable of finishing their task but we also saw that the route chosen was of worse performance than in the other networks, especially for 400 agents which registered an abysmal drop.

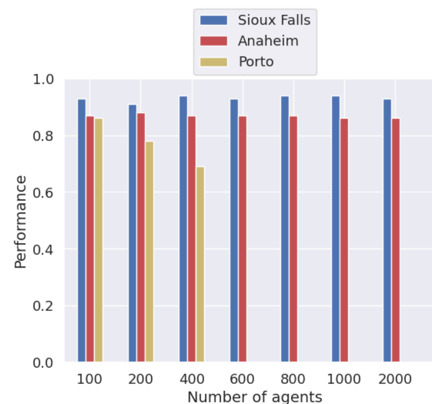

Figure 7.9: Route evaluation for Sioux-Falls, Anaheim and Porto networks

MARL approach to the Sioux Falls traffic assignment problem

The last experiment was to train a model that could assign all trips scheduled for the Sioux Falls network and its respective OD matrix [30]. The demand generated for this network includes trips between every pair of nodes, which results in a total of 360600 trips assigned. Although small, with 24 nodes and 76 links, this network is widely used in traffic assignment problems despite not being considered a realistic network.

The previously mentioned solution that is included in the repository only lists the flow and final cost for each link, meaning that if we wish to ascertain the total time spent in the network, we have to calculate for each edge of the network the time spent in it by every driver that drove through it.

Since the authors specify the usage of BPR as the volume-delay function and the use of the default parameters for α and β, we can obtain these values according to Equation 7.1.

$$Tt_a = \sum_{a=1}^{E} \left(\sum_{n=1}^{f_a} t_a * \left(1 + \alpha * \left(\frac{n}{c_a} \right)^{\beta} \right) \right) \tag{7.1}$$

where:

Tt – total time spent in the network by all drivers
E – number of links in the network
f_{a} – final flow for link a
t_{a} – free flow travel time for link a
c_{a} – predicted capacity of link a per hour

To evaluate the performance of our agents, we positioned the average time spent in the network by a driver in between an interval where the value we can obtain from Equation 7.1 divided by the number of drivers constitutes the interval's lower bound. The upper bound, and our baseline, is instead obtained by assigning the theoretical shortest path possible for all scheduled trips, running a simulation and calculating the average time spent in the network.

The tests previously done had already revealed that we could not field the number of agents required. The final solution, previously explained in Chapter 6, was to make each agent responsible for one hundred trips, which means that 3606 agents will carry the 360600 trips planned out.

The baseline approach using the shortest path results in drivers taking, on average, 46.57 minutes to complete their route. On the opposite side of the spectrum, the best-known solution for the network results in an average of 11.73 minutes for drivers to complete their trips.

We did several experiments with PPO by briefly experimenting with different hyper-parameter values but did not conduct a thorough analysis of them, as the process was very computationally intensive and time-consuming. As such, the results we provide here are based on our empirical understanding of the best hyper-parameter values for this problem.

With PPO, our agents managed to complete their trips, on average, in 19.31 minutes. While this is far from the optimal solution, it significantly improved over the baseline we established.

Also, the optimum solution pertains to a state of system optimum, which considering our approach, was unattainable, as our agents were strictly competing for the shortest time possible.

To help understand the differences between the assignments with the shortest path, reinforcement learning and the optimum solution, we produced the following figures, which depict the ratio between traffic flow and edge capacity following the rules presented previously in Subsection 6.3.3

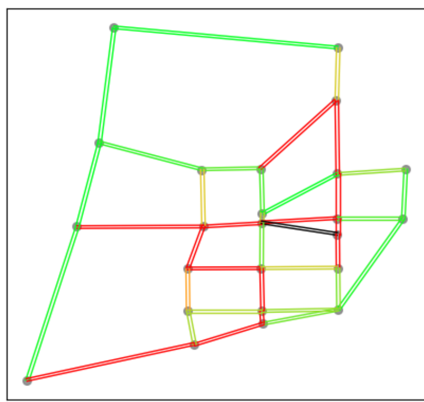

Figure 7.10: Shortest path assignment

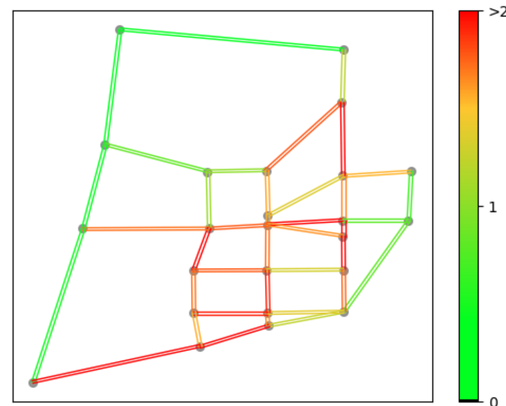

Figure 7.11: Best known solution

The differences between Figures 7.10 and 7.11 help us understand what a better assignment looks like. We are looking for an evenly spread of traffic amongst the edges, which does produce a warmer-looking graph overall, which might be counter-intuitive. Still, the most important is to minimize the number of red edges, especially one with a ratio much higher than two, which significantly impacts travel time.

The average value measured was 1.65 and 1.47, respectively, which does not convey the evident difference in performance. The critical factor, however, lies in the maximum value registered, which does differ significantly between these two approaches. The shortest path results in a maximum of 5.95, as opposed to 2.56 for the optimum solution.

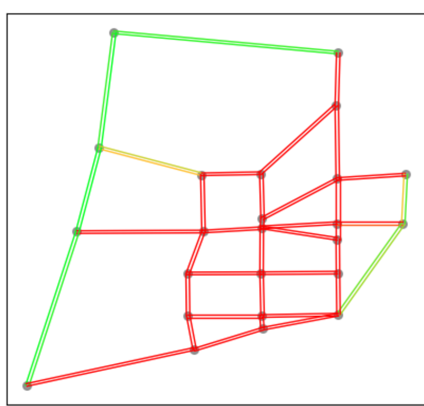

Figure 7.12: PPO exploratory behaviour

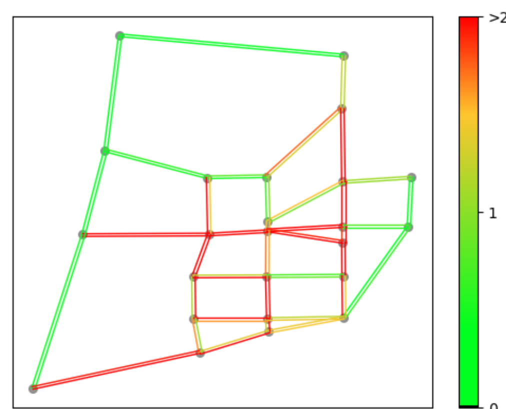

Figure 7.13: PPO final result

Our approach with PPO, which resulted in performance roughly between these two, has an average ratio of 1.58 and a maximum of 4.10. Figure 7.12 showcases the early stages of training, where agents operate primarily in an exploratory fashion, while Figure 7.13 reveals the final look of the network.

Trying to improve the performance of our agents, we tried utilizing a sigmoid function to enhance our control over the rewards coming from the simulator. We intended to further penalize agents that are taking routes that result in massive amounts of delay. The results are promising as we registered an average travel time of 18.54, a slight improvement over our definitive solution. However, training was unstable, and the solution quickly deteriorated, so we believe more experimentation is required to adjust the function correctly.

Regarding PPO, we made one additional test which consisted of increasing the number of agents to 7212, making each agent responsible for 50 trips instead of 100. This resulted in essentially the same performance as the previous test but was much more lengthy.

To conclude our experiments, we decided to run a training session with the A2C algorithm even though we expected results worse than PPO, considering the signs shown during preliminary tests. Three edges experienced traffic flow far beyond the road capacity, with the network having a maximum ratio of 8.17 and an average of 1.99. These edges were responsible for the unexpected results obtained.

With this algorithm, our agents spent, on average, 299 minutes on the network. This leads us to believe that the agents here are not valuing the reward coming from the simulation as much as we wanted, which might mean that a different reward function might be necessary to achieve better results.

Finally, we can briefly touch on some insights we collected empirically regarding the hyperparameters, specifically for the PPO algorithm applied to the Sioux Falls network. For starters, we realized that it was crucial to accurately predict the number of steps necessary for all agents to complete an entire episode.

To avoid running into episodes that might not finish or grow too long and making samples collected vary too much in size, we define an `horizon`, which is tied to the network being used. This parameter enables us to forcefully terminate the episode when all agents have performed that many steps. For the Sioux Falls network, we settled on a maximum of 15 steps allowed, roughly double the number of steps of the longest of the shortest paths between every pair of nodes, to give agents some liberty to explore the network.

With the horizon defined, we know that each episode will contain, at maximum, 15 times the number of agents present in the network, in this case, 3606, which results in a maximum of 54090 steps. Notice that as agents start to figure their routes, this value drops abruptly, as there are multiple routes with paths of size one and two.

This leads us to the `train_batch_size` parameter, which dictates the size of the sample collected from the environment or the number of steps between all agents. Since we are using `complete_episodes`, the sample will always contain at least one episode, even if the total number of steps exceeds the value set for `train_batch_size`. Alternatively, if the number of

steps is inferior, the sample will contain as many complete episodes as possible to collect that number of steps.

To promote a stable learning process, we are interested in considering multiple episodes before updating the policy function. This led us to set the `train_batch_size` parameter as twice the product of the number of agents by the `horizon` defined. This allows us to have at least two episodes in every batch during the earlier parts of training but makes it so that we do not have too many episodes per sample in the latter stages of training when episodes are getting much shorter.

Regarding the learning rate, we found 0.0005 to be the most effective value. Increasing this value led to agents converging too fast to their first solutions found, while decreasing this value resulted, in some cases, in a total lack of convergence.

As for the `lambda` parameter, which acts as a lever between bias and variance when deciding between relying more on the current estimate or the actual rewards received, we obtained better results with a value of 0.9. However, it did not seem to have much impact on the overall results.

Finally, for the `gamma` parameter, which corresponds to the discount factor of the MDP, we kept a high value of 0.95 since we want our agents to value the rewards in the future, particularly because the most important reward comes from the simulator which is only attributed in the last step.

7.3 Summary

The results show that we have managed to deliver a simulator capable of handling large-scale simulations, such as a scenario of the predicted traffic flow for a city like Porto. In addition, the MARL experiments point in a promising direction regarding using this method to capture the decision-making process that drivers practice in traffic.

Chapter 8

Conclusions and Future Work

One of the main challenges of our times is the ever-growing need for good, reliable and efficient transportation means. Answering this call requires extensive planning and investigation, as some of the adverse effects of city growth and development are known to us all by means of traffic jams, frustration and, most importantly, an increase in levels of pollution that threaten the sustainability of this process. As such, it is becoming increasingly important to plan and adapt our cities to lessen our daily commutes' environmental impact.

As the road networks of our cities keep getting more complex, deciding the best measures to improve traffic flow and reduce harmful emissions is becoming more difficult, which is why it is essential to equip researchers and planners with tools that help them understand the effects that different measures will have. This is the role of traffic simulators like the one we have implemented.

We set out to create a mesoscopic simulator capable of handling large-scale simulations independently, and that could also be used in conjunction with a multi-agent reinforcement learning system. The simulator was designed to be independent and allow simulations based on the user's input files. Additionally, it is incorporated into the machine learning module by allowing agents to create their routes in the environment and then execute them in the simulator to receive their rewards.

To tackle this challenge, we started by conducting a state-of-the-art analysis of traffic simulations, looking for current mesoscopic solutions that might match our needs concerning the integration with a multi-agent reinforcement learning system to tackle traffic assignment problems. We also researched current approaches to this problem and machine learning frameworks that would be useful in our pursuit, specifically those that provided support for multiple agents.

We tested this solution in two steps. The first was exclusively the simulator, where we conducted some stress tests to evaluate its scalability and also confirmed the capability of running a simulation of the predicted traffic for the city of Porto, based on an OD matrix, using the network of the city that was provided to the DynamiCITY project by the city council.

The second stage saw multiple experiments with the multi-agent reinforcement learning module operating the simulator. To do this, we built a small test network and used some resources

from the Transportation Networks for Research repository [19], which enabled us to compare the results our agents could achieve with the current best-known solutions obtained through optimization algorithms.

Our results show that we met the scalability requirement for the simulator, which is capable of handling simulations with millions of vehicles and that even though our results on the MARL experiments in the Sioux Falls network are still far from the optimum solution, we feel like there is still room to improve and that this method shows potential for capturing the decision-making processes of drivers in traffic.

8.1 Main Contributions

This work contributes to the current body of knowledge in this field primarily through the following topics.

- Creation of an event-driven mesoscopic simulator
- Creation of a custom OpenAI Gym environment
- Traffic assignment for the Sioux Falls network with MARL

8.2 Future Work

Looking forward, we identify two non-exclusive avenues to carry this project onward, one for each component of the solution we have proposed. The simulator can be further developed with new functionalities, while the machine learning module can be further explored and tested.

For the simulator, we see as highly valuable adding the capability of running the simulator as a service, which should drastically improve the temporal performance of the machine learning processes, as it would mean a complete removal of the setup time and of the overhead induced by launching a new sub-process every time we want to run a simulation.

As for functionalities, it would be interesting to introduce different means of transportation and the occurrence of unexpected events, such as accidents, which could temporarily reduce an edge's capacity. Finally, it's also relevant to allow the user to select different volume-delay functions for computing the travel time between edges.

As for the reinforcement learning module, the approach should involve further experimentation. Considering the potential we saw in using a sigmoid function to control better the rewards coming from the simulation, it would be interesting to try to refine it further so that it becomes more stable and experiment with other non-linear functions that might better capture the behaviour we want our agents to acquire.

Another possible approach would be to rewrite the entire reward history for all agents instead of just the last step with the information that comes from the simulator. This would undoubtedly be a challenge, but it would lead to a much more complete reward system.

Our approach used exclusively homogeneous agents, but it would be relevant to introduce different classes of drivers with different reward functions. For example, we could introduce tolling in some edges and create different profiles of drivers with different balances between the interest in saving time or money.

Finally, experimentation with other networks is welcomed, mainly since the repository mentioned contains many more networks with complete information regarding the trips and the resulting optimal solution.

References

[1] Aimsun. Aimsun next. Available at https://docs.aimsun.com/next/22.0.1/, last accessed in June 2022.

[2] Román Aragon-Gómez and Julio B. Clempner. Traffic-signal control reinforcement learning approach for continuous-time markov games. *Engineering Applications of Artificial Intelligence*, 89:103415, 2020.

[3] Jaume Barceló, Jordi Casas, David García, and Jean Marc Perarnau. Methodological notes on combining macro, meso and micro models for transportation analysis. In *11th World Conference on Transport Research*, 2007.

[4] Ana L. C. Bazzan and Ricardo Grunitzki. A multiagent reinforcement learning approach to en-route trip building. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 5288–5295, July 2016. ISSN: 2161-4407.

[5] Moshe Ben-akiva, Michel Bierlaire, Haris Koutsopoulos, and Rabi Mishalani. Dynamit: a simulation-based system for traffic prediction, 1998.

[6] Bentley. Predictive modeling and simulation of transportation. Available at https://www.bentley.com/en/products/brands/cube, last accessed in June 2022.

[7] W. Burghout, H.N. Koutsopoulos, and I. Andreasson. A discrete-event mesoscopic traffic simulation model for hybrid traffic simulation. In *2006 IEEE Intelligent Transportation Systems Conference*, pages 1102–1107, Toronto, ON, Canada, 2006. IEEE.

[8] Wilco Burghout. *Hybrid microscopic-mesoscopic traffic simulation*. Dept. of Infrastructure, Royal Institute of Technology, Stockholm, 2004. OCLC: 186617287.

[9] Dan Calderone and S. Shankar Sastry. Markov decision process routing games. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, page 273–279, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences*, 11:4948, 05 2021.

[11] Fiona Carmichael. *A guide to game theory*. Financial Times Prentice Hall, 2012.

[12] Rodrigo Castillo González, Julio B. Clempner, and Alexander S. Poznyak. Solving traffic queues at controlled-signalized intersections in continuous-time markov games. *Mathematics and Computers in Simulation*, 166:283–297, 2019.

[13] Tianshu Chu, Jie Wang, Lara Codecà, and Zhaojian Li. Multi-Agent Deep Reinforcement Learning for Large-Scale Traffic Signal Control. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):1086–1095, March 2020. Conference Name: IEEE Transactions on Intelligent Transportation Systems.

[14] Aurélien Clairais. Simulateur de trafic mésoscopique événementiel open-source. Available at https://github.com/AureClai/stream-python, last accessed in June 2022.

[15] Caliper Corporation. Transmodeler traffic simulation software. Available at https://www.caliper.com/transmodeler, last accessed in June 2022.

[16] Américo Costa and Joaquim Macedo. *Manual de Planeamento das Acessibilidades e da Gestão Viária - Níveis de Serviço em Estradas e Auto-Estradas*. Comissão de Coordenação e Desenvolvimento Regional do Norte, 12 2008.

[17] Samah El-Tantawy, Baher Abdulhai, and Hossam Abdelgawad. Multiagent Reinforcement Learning for Integrated Network of Adaptive Traffic Signal Controllers (MARLIN-ATSC): Methodology and Large-Scale Application on Downtown Toronto. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1140–1150, September 2013. Conference Name: IEEE Transactions on Intelligent Transportation Systems.

[18] US EPA. Sources of Greenhouse Gas Emissions. Available at https://www.epa.gov/ghgemissions/sources-greenhouse-gas-emissions, last accessed in June 2022.

[19] Transportation Networks for Research Core Team. Transportation networks for research. Available at https://github.com/bstabler/TransportationNetworks, last accessed in January 2023.

[20] PTV Group. Multimodal traffic simulation software. Available at https://www.myptv.com/en/mobility-software/ptv-vissim, last accessed in June 2022.

[21] Ricardo Grunitzki, Gabriel de Oliveira Ramos, and Ana Lucia Cetertich Bazzan. Individual versus difference rewards on reinforcement learning for route choice. In *2014 Brazilian Conference on Intelligent Systems*, pages 253–258, 2014.

[22] Jian Guo and Istvan Harmati. Lane-changing decision modelling in congested traffic with a game theory-based decomposition algorithm. *Engineering Applications of Artificial Intelligence*, 107:104530, 2022.

[23] Zijian Hu, Chengxiang Zhuge, and Wei Ma. Towards a Very Large Scale Traffic Simulator for Multi-Agent Reinforcement Learning Testbeds. The Hong Kong Polytechnic University, May 2021.

[24] Kuang Huang, Xu Chen, Xuan Di, and Qiang Du. Dynamic driving and routing games for autonomous vehicles on networks: A mean field game approach. *Transportation Research Part C: Emerging Technologies*, 128:103189, 2021.

[25] Ido Klein and Eran Ben-Elia. Emergence of cooperation in congested road networks using ICT and future and emerging technologies: A game-based review. *Transportation Research Part C: Emerging Technologies*, 72:10–28, November 2016.

[26] Ido Klein and Eran Ben-Elia. Emergence of cooperative route-choice: A model and experiment of compliance with system-optimal ATIS. *Transportation Research Part F: Traffic Psychology and Behaviour*, 59:348–364, November 2018.

[27] Ido Klein, Nadav Levy, and Eran Ben-Elia. An agent-based model of the emergence of cooperation and a fair and stable system optimum using ATIS on a simple road network. *Transportation Research Part C: Emerging Technologies*, 86:183–201, January 2018.

[28] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. *Computer Science Review*, 3(2):65–69, 2009.

[29] Averill M. Law. *Simulation Modeling & Analysis*. McGraw-Hill, New York, NY, USA, 5 edition, 2015.

[30] Larry J. LeBlanc, Edward K. Morlok, and William P. Pierskalla. An efficient approach to solving the road network equilibrium traffic assignment problem. *Transportation Research*, 9(5):309–318, 1975.

[31] Nadav Levy, Ido Klein, and Eran Ben-Elia. Emergence of cooperation and a fair system optimum in road networks: A game-theoretic and agent-based modelling approach. *Research in Transportation Economics*, 68:46–55, August 2018.

[32] Minne Li, Zhiwei Qin, Yan Jiao, Yaodong Yang, Jun Wang, Chenxi Wang, Guobin Wu, and Jieping Ye. Efficient Ridesharing Order Dispatching with Mean Field Multi-Agent Reinforcement Learning. In *The World Wide Web Conference*, WWW '19, pages 983–994, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[34] Kaixiang Lin, Renyu Zhao, Zhe Xu, and Jiayu Zhou. Efficient Large-Scale Fleet Management via Multi-Agent Deep Reinforcement Learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1774–1783, London United Kingdom, July 2018. ACM.

[35] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 157–163. Morgan Kaufmann, San Francisco (CA), 1994.

[36] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[37] Sven Maerivoet and Bart De Moor. Transportation planning and traffic flow models. *arXiv: Physics and Society*, 2005.

[38] Chao Mao and Zuojun Shen. A reinforcement learning framework for the adaptive routing problem in stochastic time-dependent network. *Transportation Research Part C: Emerging Technologies*, 93:179–197, August 2018.

[39] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. JMLR.org, 2016.

[40] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36 1:48–9, 1950.

[41] Robert Neuhold and Martin Fellendorf. Volume delay functions based on stochastic capacity. *Transportation Research Record: Journal of the Transportation Research Board*, 2421:93–102, 12 2014.

[42] Ann Nowé, Peter Vrancx, and Yann-Michaël De Hauwere. *Game Theory and Multi-agent Reinforcement Learning*, pages 441–470. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[43] United States. Bureau of Public Roads. *Traffic Assignment Manual for Application with a Large, High Speed Computer*. Number vol. 2 in Traffic Assignment Manual for Application with a Large, High Speed Computer. U.S. Department of Commerce, Bureau of Public Roads, Office of Planning, Urban Planning Division, 1964.

[44] The International Council on Clean Transportation. Transport could burn up the EU's entire carbon budget. Available at https://theicct.org/transport-could-burn-up-the-eus-entire-carbon-budget/, last accessed in June 2022.

[45] Andreas Pell, Andreas Meingast, and Oliver Schauer. Trends in Real-time Traffic Simulation. *Transportation Research Procedia*, 25:1477–1484, December 2017.

[46] Arnu Pretorius, Kale ab Tessera, Andries P. Smit, Kevin Eloff, Claude Formanek, St John Grimbly, Siphelele Danisa, Lawrence Francis, Jonathan Shock, Herman Kamper, Willie Brink, Herman Engelbrecht, Alexandre Laterre, and Karim Beguir. Mava: A research framework for distributed multi-agent reinforcement learning. *arXiv preprint arXiv:2107.01460*, 2021.

[47] Gabriel de O. Ramos, Ana L. C. Bazzan, and Bruno C. da Silva. Analysing the impact of travel information for minimising the regret of route choice. *Transportation Research Part C: Emerging Technologies*, 88:257–271, March 2018.

[48] Robert W. Rosenthal. A class of games possessing pure-strategy nash equilibria. *Int. J. Game Theory*, 2(1):65–67, dec 1973.

[49] Mustapha Saidallah, Abdeslam El Fergougui, and Abdelbaki El Belrhiti El Alaoui. A Comparative Study of Urban Road Traffic Simulators. *MATEC Web of Conferences*, 81:05002, January 2016.

[50] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[51] Ammar Saric, Sanjin Albinovic, Suada Dzebo, and Mirza Pozder. Volume-delay functions: A review. In Samir Avdaković, editor, *Advanced Technologies, Systems, and Applications III*, pages 3–12, Cham, 2019. Springer International Publishing.

[52] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[53] Muhammad Sameer Sheikh, Ji Wang, and Amelia Regan. A game theory-based controller approach for identifying incidents caused by aberrant lane changing behavior. *Physica A: Statistical Mechanics and its Applications*, 580:126162, 2021.

[54] Masoud Soleymani Shishvan and Jörg Benndorf. Operational decision support for material management in continuous mining systems: From simulation concept to practical full-scale implementations. *Minerals*, 7(7), 2017.

[55] Zhenyu Shou, Xu Chen, Yongjie Fu, and Xuan Di. Multi-agent reinforcement learning for Markov routing games: A new modeling paradigm for dynamic traffic assignment. *Transportation Research Part C: Emerging Technologies*, 137:103560, April 2022.

[56] Zhenyu Shou and Xuan Di. Reward Design for Driver Repositioning Using Multi-Agent Reinforcement Learning. *Transportation Research Part C: Emerging Technologies*, 119:102738, October 2020. arXiv:2002.06723 [cs, stat].

[57] L. Smith, R. Beckman, and K. Baggerly. TRANSIMS: Transportation analysis and simulation system. Technical Report LA-UR-95-1641, Los Alamos National Lab. (LANL), Los Alamos, NM (United States), July 1995.

[58] Fernando Stefanello, B. B. D. Silva, and Ana Lúcia Cetertich Bazzan. Using topological statistics to bias and accelerate route choice: Preliminary findings in synthetic and real-world road networks. In *ATT@IJCAI*, 2016.

[59] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[60] Takashi Tanaka, Ehsan Nekouei, Ali Reza Pedram, and Karl Johansson. Linearly solvable mean-field traffic routing games. *IEEE Transactions on Automatic Control*, PP:1–1, 04 2020.

[61] J. K Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sulivan, Luis Santos, Rodrigo Perez, Caroline Horsch, Clemens Dieffendahl, Niall L Williams, Yashas Lokesh, Ryan Sullivan, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning. *arXiv preprint arXiv:2009.14471*, 2020.

[62] Jordan K Terry, Benjamin Black, and Mario Jayakumar. Magent, 2020. Available at `https://github.com/Farama-Foundation/MAgent`, last accessed in November 2022.

[63] Xiaoqiang Wang, Liangjun Ke, Zhimin Qiao, and Xinghua Chai. Large-Scale Traffic Signal Control Using a Novel Multiagent Reinforcement Learning. *IEEE Transactions on Cybernetics*, 51(1):174–187, January 2021. Conference Name: IEEE Transactions on Cybernetics.

[64] Yi Wang, W. Y. Szeto, Ke Han, and Terry L. Friesz. Dynamic traffic assignment: A review of the methodological advances for environmentally sustainable road transportation applications. *Transportation Research Part B: Methodological*, 111:370–394, May 2018.

[65] Zhe Xu, Zhixin Li, Qingwen Guan, Dingshui Zhang, Qiang Li, Junxiao Nan, Chunyang Liu, Wei Bian, and Jieping Ye. Large-Scale Order Dispatch in On-Demand Ride-Hailing Platforms: A Learning and Planning Approach. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 905–913, London United Kingdom, July 2018. ACM.

[66] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. Mean Field Multi-Agent Reinforcement Learning, December 2020. Number: arXiv:1802.05438 arXiv:1802.05438 [cs].

[67] Nadezda Zenina and Yuri Merkuryev. The basis for the transportation microscopic simulation model validation. *2019 60th International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS)*, pages 1–6, 2019.

[68] Huichu Zhang, Siyuan Feng, Chang Liu, Yaoyao Ding, Yichen Zhu, Zihan Zhou, Weinan Zhang, Yong Yu, Haiming Jin, and Zhenhui Li. CityFlow: A Multi-Agent Reinforcement Learning Environment for Large Scale City Traffic Scenario. In *The World Wide Web Conference*, pages 3620–3624, San Francisco CA USA, May 2019. ACM.

[69] Bo Zhou, Qiankun Song, Zhenjiang Zhao, and Tangzhi Liu. A reinforcement learning scheme for the equilibrium of the in-vehicle route choice problem based on congestion game. *Applied Mathematics and Computation*, 371:124895, April 2020.

[70] Ming Zhou, Jiarui Jin, Weinan Zhang, Zhiwei Qin, Yan Jiao, Chenxi Wang, Guobin Wu, Yong Yu, and Jieping Ye. Multi-Agent Reinforcement Learning for Order-dispatching via Order-Vehicle Distribution Matching. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2645–2653, Beijing China, November 2019. ACM.