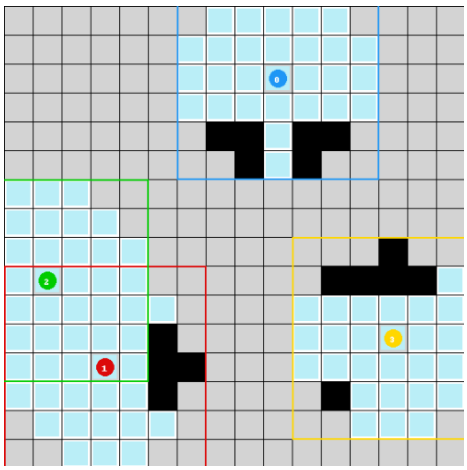
TÉCNICO LISBOA



IndoorExplorers: an OpenAI Gym environment for Multi-UAV Exploration Algorithms

Alexandra Isabel Fernandes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. António Manuel Raminhos Cordeiro Grilo
Prof. João Paulo Carvalho

Examination Committee

Chairperson: Prof. Pedro Filipe Zeferino Aidos Tomás
Supervisor: Prof. António Manuel Raminhos Cordeiro Grilo
Member of the Committee: Prof. Alberto Manuel Martinho Vale

November 2023

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

This work was created using LaTeX typesetting language in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my family for caring and supporting over all these years, for always being there for me through thick and thin.

I would like to show my deepest gratitude to all the medical team from Hospital dos Capuchos and health care professionals that accompanied me during my recovery.

I would also like to acknowledge my dissertation supervisors Prof. António Manuel Raminhos Cordeiro Grilo and Prof. João Paulo Carvalho for their insight, support and sharing of knowledge that has made this thesis possible.

To all my friends and colleagues that helped me grow as a person and have accompanied me through my academic years. Thank you for all the memories and support.

Last but not least, to my girlfriend, thank you for always being there, during the good and bad times, specially when there is a mess in my head and I wanted to give up. Thank you for listening, encouraging me and for all the love and support you have shown me.

To each and every one of you – Thank you.

Abstract

The goal of this work was to create an OpenAI Gym environment to simulate indoor exploration scenarios by a swarm of autonomous Unmanned Aerial Vehicles (UAVs), each equipped with Light Detection And Ranging (LiDAR) sensors and with safe flying capabilities, including the detection and avoidance of any objects, across the space in question. The exploration tasks consists in determining the optimal path that gathers as much information about the space as possible, in this case to create a map of the space. Using a swarm of UAVs, it is possible to achieve these tasks faster, with fewer costs and safely for humans.

The developed OpenAI Gym-based environment was then used to test a Reinforcement Learning (RL) algorithm for path planning, specifically Dueling Double Deep Q-Learning (DDDQN). The developed environment currently allows tests in 2D maps with up to four UAVs equipped with a simplified simulated LiDAR sensor, with or without communications.

The results obtained compare two approaches to accelerate the training of the DDDQN. Furthermore, an analysis of the impact of more than one agent and whether communications affect the performance was done.

Keywords

Indoor Exploration, UAV Swarm, DRL, OpenAI Gym

Resumo

O objetivo deste trabalho é o desenvolvimento de um ambiente de simulação de OpenAI Gym para a exploração de um espaço interior, com recurso a um enxame de Unmanned Aerial Vehicles (UAVs), Aeronaves Não Tripuladas em português. Cada UAV estará equipado com sensores Light Detection And Ranging (LiDAR) e deverá ter a capacidade de navegar de forma segura pelo espaço em questão, ou seja, deverá ser capaz de detetar e evitar qualquer tipo de objetos. A exploração de um espaço consiste em planear um caminho ótimo para recolher o máximo de informação possível sobre o mesmo, neste caso para a criação do seu mapa. O recurso a enxames de UAVs permite a concretização de tarefas mais complexas, rapidamente, com menos custos e de forma mais segura para as pessoas.

O ambiente foi desenhado para ser compatível com a infraestrutura de OpenAI Gym e foi utilizado para testar um algoritmo de Reinforcement Learning (RL) para o planeamento de trajetórias, concretamente Dueling Double Deep Q-Learning (DDDQN). De momento, o ambiente desenvolvido permite realizar testes em mapas 2D com até quatro UAVs, cada um equipado com um sensor LiDAR simulado, em cenários com e sem comunicação entre UAVs.

Os resultados obtidos comparam duas abordagens para acelerar o treino da DDDQN. Adicionalmente, também foi feita uma análise do impacto que o número de agentes tem e como é que a comunicação afeta a performance do algoritmo .

Palavras Chave

Exploração de espaços interiores, Enxame de UAVs, DRL, OpenAI Gym

Contents

List of Figures

List of Tables

Listings

Acronyms

A3C Asynchronous Advantage Actor Critic

ACO Ant colony optimization

AEC Agent Environment Cycle

CPP Coverage Path Planning

DFS Depth First Search

DQN Deep Q-Learning

DDDQN Dueling Double Deep Q-Learning

DRL Deep Reinforcement Learning

FANET Flying Ad-hoc Network

FoV Field of View

GBS Ground Base Station

GNSS Global Navigation Satellite System

GPS Global Positioning System

GSO Glow-worm Swarm Optimization

IMU Inertial Measurement Unit

LiDAR Light Detection And Ranging

MAV Micro Aerial Vehicle

MDP Markov Decision Process

POSG Partially Observable Stochastic Game

POMDP Partially Observable Markov Decision Process

PRM Probabilistic Road Maps

PRM* Probabilistic Road Maps Star

PSO Particle Swarm Optimization

PPO Proximal Policy Optimization

RL Reinforcement Learning

ROS Robot Operating System

RRT Rapidly exploring Random Trees

RRT* Rapidly exploring Random Trees Star

SAC Soft Actor-Critic

SI Swarm Intelligence

SITL Software In-The-Loop

SLAM Simultaneous Localization and Mapping

UAV Unmanned Aerial Vehicle

1

Introduction

Contents

Unmanned Aerial Vehicles (UAVs), as the name suggests, are airborne devices that do not have any onboard crew or passengers, meaning that they can be controlled remotely or perform autonomous tasks. UAVs can be fixed wing or rotary wings, in the latter case they can be called drones. The predecessors of the current UAVs were developed during the First World War, being used for military reconnaissance missions or as missiles. Nowadays, these devices have a broad diversity of applications that range from security and surveillance [7], cinematographic filming [8], disaster management (which includes search and rescue operations or response to natural disasters) [9], monitoring of specific areas or infrastructure (such as borders, power lines, dams, etc.) [10], agriculture [11], to name a few.

The increasing demand of UAV technology is due to their potential to be faster, more efficient and safer than using human labour and putting human lives at risk for some of the applications mentioned above. They can vary in weight, range, size, configuration, payload, engine type and performance characteristics. According to these attributes, it is possible to customise what sensors, cameras, communication protocol and navigation methods to use according to the application in mind. [15]

UAVs can work as a single unit or in coordinated groups. The latter option presents benefits such as dividing the workload among all units, being faster and able to accomplish larger and more complex tasks. One way to achieve this coordinated behaviour is to look at how nature works, for instance, how groups of animals share information between each member of the group and coordinate in order to achieve the greater reward for the whole group. This happens with birds for example, the whole flock communicates their findings between each other to get sustenance and provisions. This is what some Nature Inspired Algorithms are designed upon, more specifically what Swarm Intelligence (SI) algorithms are all about.

A swarm consists of a coordinated group of entities that communicate with one another, sharing crucial information and working together towards a common goal. SI algorithms with UAVs are commonly used for path planning or exploration tasks.

The goal of the exploration task consists in gathering as much information about the space as possible in an optimal way. Concurrently, with the information gathered, it is possible to combine this with a mapping task when it comes to unknown environments. There is a wide range of spaces that can be explored, such as indoor or outdoor, static or dynamic environments, or even a mix of the two. Each type of space has unique challenges to be faced, for instance, an indoor space does not have the influence of weather conditions that affect the UAV's flight manoeuvrability, but on the other hand it is deprived of any Global Navigation Satellite System (GNSS) such as Global Positioning System (GPS), so other localization methods are needed. As for the difference between static and dynamic spaces, it relies on the existence or non-existence of moving environmental elements, such as any obstacles like a person walking by or a random object falling.

Autonomous exploration grants the ability to achieve tasks in a more efficient way and without risking

human lives in hazardous environments and in emergency situations.

Focusing on indoor dynamic environments, autonomous exploration using UAVs is useful for surveillance purposes, for instance finding an intruder in a household, or in search-and-rescue scenarios, where UAVs are able to pass through crevices and the middle of debris, in a more agile and faster way than any human could.

In most exploration scenarios, a group of UAVs reveals to perform better than a single UAV [22]. As explained above, a swarm of UAVs is capable of jointly and efficiently performing the tasks mentioned above. In addition, they can cover larger and more complex spaces faster than a single UAV. Although the term SI was first defined in [14], its popularity has recently increased, especially with UAV applications that require the coverage of large areas or the division of tasks. There are several levels of autonomy and different UAV swarm communication and control architectures that are described in [16]. In simpler terms, there are three types of possible approaches and then some variations in each, they differ in the way communication and organization is processed:

1. **Centralized system** - There is a Ground Base Station (GBS) that receives information (data from sensors and other peripherals) from each UAV and then sends back updates on how each one should operate.

2. **Decentralized system** - Each UAV works independently and communicates with the other UAVs. By sharing information, they can coordinate how each one should act.

3. **Hybrid system** - The GBS can divide the space under analysis into smaller areas and then assign different UAVs to each. The UAVs would then organize themselves within each area.

In centralized and hybrid systems, GBS is usually a computer that performs all or most of the computational processing necessary. In the case of a decentralized system, all processing must be done on board, which presents challenges with respect to the specifications of the real-life UAV and what it is able to carry without compromising its integrity (the UAV's payload) or how long does its battery last. This means when having a specific application in mind, there must be a compromise between the UAV's characteristics, and consequently its capabilities, and the sensors and peripherals necessary for the task. Most of the sensors and peripherals used are stereo cameras,Inertial Measurement Unit (IMU) , ultrasound sensors, Light Detection And Ranging (LiDAR), among others. [12]

One problem that occurs is maintaining the communication between nodes, in this case UAVs, they can communicate only with their closest neighbours and information should be spread to all nodes. For this purpose, Flying Ad-hoc Networks (FANETs) are useful, where each UAV may work as a router for others, maintaining the network connected by adjusting their positions, furthermore there needs to be at least one UAV connected to the GBS [13].

Having this context in mind, a suited simulation environment was necessary to simulate the intended behaviour. Given the emergence of the topic, there are a few existing solutions but each with its own specifications, such as having integration with a standardised Reinforcement Learning (RL) platform such as OpenAI Gym, being suited for single-agent or multi-agent training, scalability, compatibility and so on (all these specification will be further analysed in Chapter 2). In the midst of diversity there was no standardised simple solution that could accommodate all the features intended, specifically a solution that allowed multi-agent Deep Reinforcement Learning (DRL) training, with communications in GNSS-denied unknown environments, which is the goal of this project.

1.1 Objective

The scope of this work consists of creating a simulation environment that allows the development and testing of RL solutions for indoor exploration problems, using an autonomous UAV swarm equipped with emulated LiDAR sensors - that is able to fly safely, including the detection and avoidance of any objects, across the space in question.

The environment aims to simulate a decentralized system of UAVs, where each UAV will communicate with its peers and process its own acquired data in order to achieve the task in hand. The developed environment can be found in [101].

A specific DRL algorithm will be tested in the developed environment, the Dueling Double Deep Q-Learning (DDDQN), which will be described in more detail in Chapter 2 and Chapter 3.

1.2 Document outline

The rest of the thesis is structured as follows:

- **Section 2:** presents the intended architecture to simulate on the developed environment. Finalizing with an overview of the existing simulation environments and software accompanied by their features;
- **Section 3:** describes how the presented objective was achieved, by presenting the developed OpenAI gym environment, according to some assumptions and the simulation setup available and exposing its limitations and features;
- **Section 4:** summarizes the obtained results accompanied by a brief analysis;
- **Section 5:** presents the conclusions and possible further developments.

2

Background and Related work

Contents

In this chapter, the architecture of the target system for each UAV is presented, exploring how each component could be integrated in the simulated environment and possibly in a real life implementation. Specifically, which sensors would be used to acquire data, how to perform localization and mapping and so on. Moreover, at the end of the chapter, an extensive survey is done regarding existing simulation environments and other relevant packages and frameworks that could accommodate the requirements for the desired architecture and the scenario previously exposed.

2.1 Architecture

Exploration problems are usually separated in several components depicted in Figure 2.2

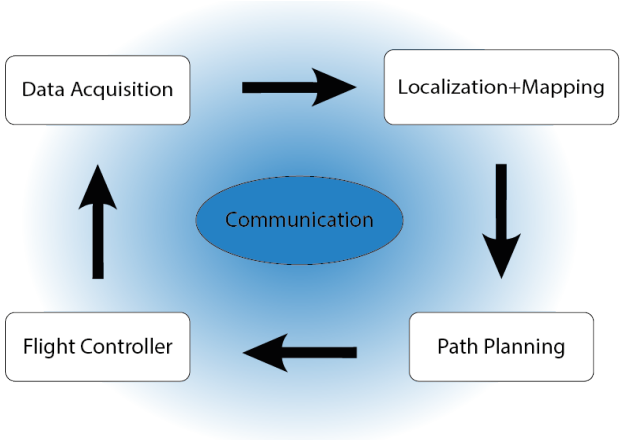


Figure 2.1: Brief architecture scheme

In a general exploration scenario, each robot or UAV must be able to do the following:

1. **Data Acquisition** In order to identify obstacles and the current location, the sensors and peripherals chosen gather information about the environment. The data gathered may be of different types, such as images or video (using cameras), distance measurements (LiDAR or SONAR), etc;

2. **Localization and Mapping** - For this Simultaneous Localization and Mapping (SLAM) algorithms are used. In order to be able to explore an unknown place, it is necessary to know with some degree of certainty the current location. This step is important to plan the next path to take;

3. **Communication and Coordination** - Since the goal is to use multiple UAVs, they must be able to communicate, in order to coordinate their movements for more efficient exploration;

4. **Path Planning** - With the gathered information from the sensors and other UAVs, each UAV can then plan its next movement;

5. **Path Following** - After the path is set, the flight controller will move according to what was planned and taking into consideration possible obstacles;

It is important to denote that the arrows in Figure 2.2 are merely indicative of the data flow and the frequency at which some of these tasks is done is not the same. For example, data is constantly being acquired (through the sensors and communication channels), so the localization and mapping will be done with the available data at the time and so forth, path planning is then done using the data available from the localization and mapping at the time it is requested.

According to the initial choice of simulation environment, the simulation framework would be similar to the one found in [1] and depicted in Figure 2.2.

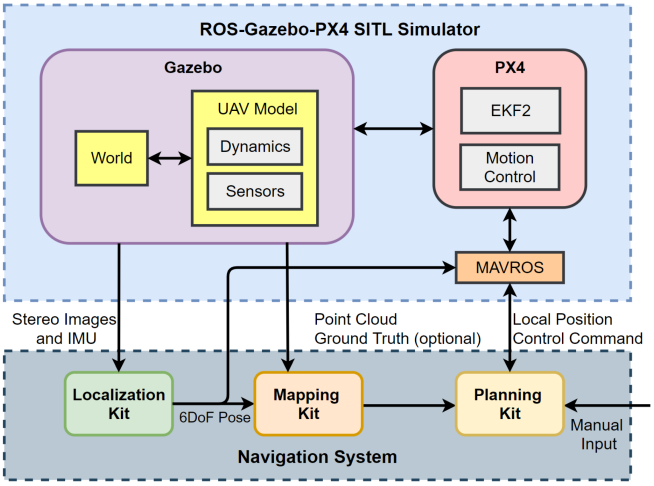


Figure 2.2: Framework of simulation from [1]

2.1.1 Data acquisition

Initially the author seeked to find the most reliable data source and some research was done to figure out which LiDAR would fit best, for simulation and having in mind future possible real tests. Alongside the sensor choice, a brief analysis of which data storage method would be more efficient. Even though in the development of the present project the focus shifted to a simpler approach, this information might be important to consider for future developments and enhancements.

2.1.1.A Choice of sensors

In order to gather information about the environment and the status of each UAV a set of sensors are necessary. It is important to know the pose of each UAV on the map and if there are any changes to the map, given that in a first instance, it will be considered that the shape and size of the area will be provided a priori.

Firstly, that are two main approaches related to mapping and localization in GNSS-denied environments: LiDAR-based and vision-based (or photogrammetry). The first option relies on laser beams to measure distances and create images based on point cloud maps. Meanwhile, the second one relies on images captured from cameras. In [24] it is possible to check some of the available LiDAR scanners for airborne systems in the market and get an insight about the variety and specifications of these scanners.

According to the information found on [25], relatively to the pros and cons of using LiDAR and photogrammetry, in this case, the photo-realistic maps provided by photogrammetry are not necessary (such as texture information or real images). But instead, it is important to get the general contour of obstacles to have safe navigation, maybe for search and rescue operations photogrammetry may be useful jointly with computer vision technology to identify who or what is being rescued.

In [43] , a solution is proposed to the problems associated with each of these two main approaches. On one hand using a rotary 3D LiDAR, whose vibrations affect the result or may even damage the scanner. On the other hand, with the vision based approach, the result has relatively lower mapping resolution when compared to LiDAR based mapping, due to the limited image pixels and other technical difficulties. They tackle the problems by using 100% solid-state LiDAR (SSL) and test them in different configurations. They conclude that LiDAR approaches are more mature and solutions, while vision-based ones seem to be the direction of research for the future.

According to [1], the authors state in two short lines the pros and cons of photogrammetry and LiDAR-based approaches: "The LiDAR sensors have a wide detection range and can directly provide high-precision depth information. However, they are also expensive, heavy, and large, which limits the application scenarios of LiDAR on the UAV platform. In contrast, the camera has simple structure, light weight, and cheap price". In their work, they provide a simulation platform with all of the localization, mapping, and path-planning kits in one simulator.

To conclude, a LiDAR approach was chosen, given that in this case there is no cost associated to using it. If a real prototype was to be built, the use of a camera would have a better cost-efficiency ratio. A camera has a lower cost and with the correct data processing achieves similar results. For simplicity and since it is not the focus of this work, it was chosen a 3D LiDAR that provides an omnidirectional Field of View (FoV) jointly with a IMU to know the pose of each UAV. Even though vision-based approaches will be the future, the author chose to use a mature and solid know approach. In addiction, 3D LiDAR is well supported by the initial chosen simulation environment, which will be mentioned further.

2.1.1.B Choice of data storage/type of map

In addition to the sensors needed to gather information, it is necessary to find how the map data should be stored.

According to [1], there are three types of maps (ways to store information about the environment)

used in the context of UAV navigation:

1. **Point cloud map** - the data obtained directly from the sensor.
2. **Occupancy map or OctoMap** - This method would be a good choice since the size of the map is known.
3. **Euclidean Signed Distance Fields (ESDFs) map** - this method is useful when there is a dynamically growing map. Since it is considered that the size of the area to explore is know, this method does not make sense for this work.

Since a simpler approach was taken, there was no need to choose amongst these three options. But the author thought this information should be coupled with the choice of sensors.

2.1.2 SLAM (Simultaneous Localization and Mapping)

SLAM algorithms based on images obtained from cameras will be discarded for simplicity, only solutions around LiDAR sensors are being considered. Given that a Robot Operating System (ROS)-based solution was in order, some toolkits and resources were object of study, having found the following options:

- gmapping [27]
- hector-slam [45]
- slam_toolbox [46]

As explained in [46], slam_toolbox seeks to bridge the gaps found in previously used packages and furthermore mentions two other packages (Cartographer [47] and KartoSLAM [28]) and states that given their complexity it is hard to modify them, thus making them unfit for many applications. Compared to gmapping and hector-slam, on one hand, hector-slam is unsuitable for reliable mapping of large spaces or when using laser scanners with low update rates. On another hand, gmapping is also not well suited for large spaces and fails to accurately close loops at an industrial scale. Additionally, since it is a filter-based approach, it cannot be easily reinitialized across multiple sessions. To this extend, slam_toolbox is the best solution, providing an approach for multi-session mapping and localization at industry scale.

Once again, even though this information was not used in the final product, it was part of the research process and it is succinctly mentioned since it might be useful for future projects.

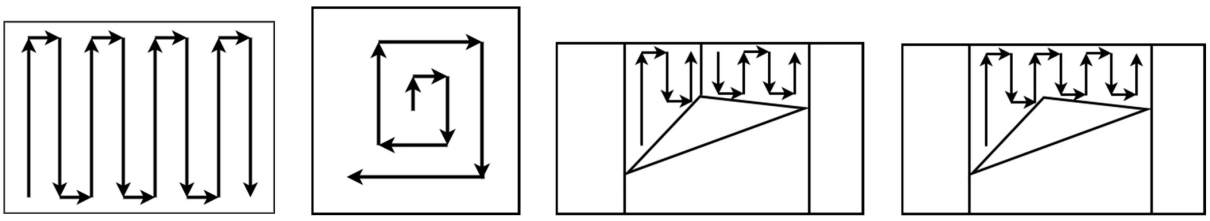
2.1.3 Path Planning

The exploration problem may be also seen as a Coverage Path Planning (CPP) problem, where the objective is to plan the optimal path to cover a determined area or space.

In [2], a survey of coverage path planning is presented according to the decomposition methods, such as no decomposition, exact, and approximate decomposition methods. It addresses some offline and online algorithms, to clarify:

- **Offline algorithms** - plan a path with prior information of the area;
- **Online algorithms** - use the real-time data retrieved from onboard sensors about the environment area to plan the path.

Some examples of decomposition methods can be found in Figure 2.3 to illustrate some of the simplest type of patterns that exist.



(a) Boustrophedon pattern **(b)** Square pattern **(c)** Trapezoidal decomposition **(d)** Boustrophedon decomposition

Figure 2.3: Some decomposition methods, images from [2]

After some research and reading (some more references are given in the bibliography, even though they were not mentioned), the author came to the conclusion that CPP solutions might not be quite that similar to what is intended with the proposed work. CPP methods rely on cellular decomposition methods and are more focused in covering the whole area, but in exploration problems, what is more important is collecting information about it. In this line of thought, these solutions were disregarded.

A comparison of several path planning strategies for autonomous exploration and mapping of unknown environments was examined in [22]. Its authors concluded that the selection of the best exploration algorithm varies according to the application in mind. Furthermore, some requirements, such as the exploration time and the map's quality and robustness may conflict with each other, by improving one of them, the other may worsen.

The two main approaches to the exploration and optimal path planning problem: (a) Frontier-based algorithms; and (b) Sampling-based algorithms. A brief explanation of each will be provided, alongside with some examples of work developed where these were used.

2.1.3.A Frontier-based methods

The concept of this type of algorithm was first introduced in [17] and, as the name suggests, relies on a frontier. A frontier distinguishes the boundary line between the explored and unexplored regions on the map. As the robot or UAV navigates, the gathered information is always increasing, making the frontier advance further. The exploration process comes to an end when there are no more frontiers to analyze.

For instance in [20], a single UAV is capable of autonomously exploring and mapping an unknown environment, without any prior information about it. It is inferred that to store the information about the 3D space's cubicle voxels representation, an Octree structure is better than normal occupancy grids, since (a) the access time is smaller; and (b) it is memory efficient since the map size is smaller.

2.1.3.B Sampling-based methods

Points are sampled from the environment and there is a weight associated with each. The robot or UAV will move to the point with the highest weight at that time. This weight is associated with a cost function that is determined by several factors, such as the expected area to be covered while moving to that point, the distance to it, and other factors.

In [18] , popular sampling-based algorithms, such as Rapidly exploring Random Trees (RRT) and Probabilistic Road Maps (PRM), are analyzed according to the quality of the solutions they find. They prove that most cannot find an optimal solution, even with infinite samples. On the counterpart, they provide efficient versions of the latter algorithms (RRT* and PRM*) that are asymptotically optimal, which means "they will return a solution to the path planning problem with high probability if one exists, but the cost of the solution returned by the algorithm will not converge to the optimal cost as the number of samples increases". In [19], J. D. Gammell and M. P. Strub continued their work in a more extensive way, achieving the same results for these algorithms.

2.1.3.C Swarm Intelligence (SI)

Nature inspired algorithms mimic the behaviour of living organisms or their corresponding communities, if it is the case. SI is an example of one type of these algorithms that focuses on a group's behaviour to coordinate efforts.

Even though there are several SI algorithms, Particle Swarm Optimization (PSO) appears to be a good option given the context of this work, which is indoor autonomous exploration by a swarm of UAVs. [33] In this reference, a comparison of several SI algorithms was done, outlining their purpose, advantages and disadvantages. Other SI algorithms included are Glow-worm Swarm Optimization (GSO) or Ant colony optimization (ACO).

In [34]'s study, it is concluded that using the PSO algorithm is a viable approach to multi-agent

navigation in dynamic environments, as the basis of a path finding algorithm. They propose a PSO path finding algorithm, jointly with Drone Flock Control (DFC) to model several modules for a controller for systems of agent in 3D environments, minimizing collisions.

After careful consideration, it can be stated that PSO is a good approach for navigation problems with one defined destination point. Since in PSO, one particle would correspond to one possible trajectory for each UAV and then all particles would converge to the defined point, resulting in a single trajectory that minimizes the desired cost function (having into account the chosen factors, for example battery or fuel consumption, time and distance covered, etc). In the context of this work, it would not be feasible, given that an online approach was necessary. For every time each UAV needs to adjust its trajectory with the new information it finds (from its sensors and information gathered from other UAVs), it would not the computationally feasible and very time consuming to have PSO compute a new route for each.

Additionally, two other nature algorithms are presented given their relevance, even though there are several more. Firstly, one interesting algorithm found was a variation of the original bug algorithm, the Swarm Gradient Bug Algorithm (SGBA), that was developed in the thesis [21]. The author states that in an indoor exploration scenario, Micro Aerial Vehicles (MAVs) are preferred, since they are safer to use near people and are also cheaper to replace in the event of a collision. In this work it was possible to have up to 6 MAVs navigating completely autonomously in a multi-room exploration scenario and return to their initial position. Its main application was search and rescue missions, so no external positioning system or outside processing were available. All of the sensing and processing must be on-board and the pocket drones need to have direct inter-drone communication. An original version of a bug algorithm was developed, which does not over-rely on perfect location as other bug algorithms do. With SGBA, the pocket drones can avoid each other based on the signal strength of the intra-drone communication, and can also locally coordinate their search based on their transmitted preferred exploration direction.

Secondly, another interesting approach is Bat algorithms, that was just briefly explored. In [40], an Improved Bat Algorithm (IBA) is proposed that integrates several ideas from bee colonies and that in some of the exploration scenarios designated by the authors had better results than PSO.

2.1.3.D Deep Reinforcement Learning (DRL)

In general terms, Reinforcement Learning (RL) consists on having an agent which interacts with an environment, in the context of this work that would be each UAV and the conjunction of the real environment with the other UAVs of the swarm, respectively. The agent affects the environment through actions, and the environment responds to those actions, giving back its state and a reward value that evaluates how well the action contributed to the overall goal of the agent. On the other hand, DRL has the same principle but uses a neural network that is trained to improve the function that maximizes rewards, having

as inputs the information about the state and reward and as output an action. Through the established feedback loop, it is possible to keep training the neural network.
The solution that is going to be explored is DDDQN, but to explain it, it is necessary to first explain its predecessor, the Deep Q-Learning (DQN). DQN is based on the Q-learning algorithm, a classic RL technique. Q-learning is used for estimating optimal action-selection policies in a Markov Decision Process (MDP) when the rewards are unknown initially.

A side note about MDP, basically it provides a mathematical framework for solving RL problems. Almost all RL problems can be modelled as an MDP or Partially Observable Markov Decision Process (POMDP). Any MDP can be defined by five variables:

- S - a set of states
- A - a set of actions
- R - a reward function
- ρ - a transition function
- γ - a discounting factor

There is also a policy π, that is learned and defines the agent's behaviour in an environment. It defines the next action to be taken according to a certain state.

There are also value functions that determine what is good for the agent in the long run, unlike the immediate reward (R). There are two types of value functions:

- **V(s)** - maps value to each state, measuring how good being in each state(s) is;
- **Q(s,a)** - maps each action to a value, measures how good a certain action(a) is, given a certain state(s).

On the other hand, a POMDP [48] models an agent's decision process in which it is assumed that the agent cannot directly observe the underlying state, even though the dynamics are still determined by an MDP, for instance, there are sensor uncertainties or the environment is not entirely visible.

Advancing onto Deep Q-Learning (DQN), it was first introduced in [49] and it consists of a neural network architecture that learns to make decisions by estimating the quality (Q-value) of actions in a given state. In the context of RL, it's used to train agents on how to take actions in an environment to maximize a reward. As stated in [3], DQN is suitable for problems with limited states and discrete action space, meanwhile DDDQN is suitable for unlimited states and discrete action space - which is appropriate for the exploration problem in unknown environments, given the uncertain nature of the unknown environment it is not possible to quantify the number of states, but the available action space is well defined and discrete, for example in can be composed of movements such as left, right, up or down. Additionally, AlMahamid *et al.* explains how DDDQN appeared to solve problems in its previous versions, the Double DQN and Dueling DQN.

The Double DQN uses two networks to solve the overestimation problem found in DQN, the Policy Network and the Target Network. The first one, optimizes the Q-value, and the second one is a replica of the first network, and it is used to produce the estimated Q-value. [50] The target network parameters are updated after a certain number of time steps by copying the weights of the policy network.

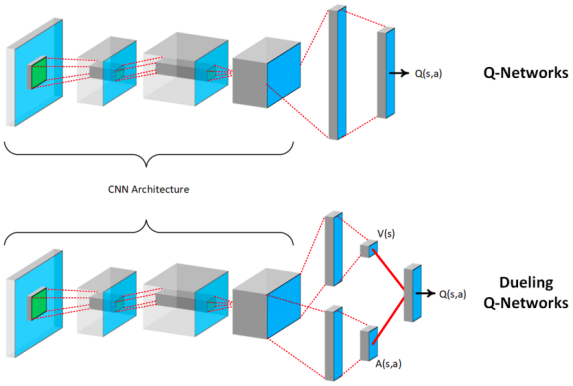


Figure 2.4: DQN vs. Dueling DQN, image from [3]

Dueling DQN decomposes the Q-value function into two functions, in an attempt to better evaluate the Q-value (also seen in Figure 2.4):

- **State-Value function V(s)** measures how good is for the agent to be in state s;
- **Advantage-Value function A(s, a)** captures how good is an action compared to other actions at a given state.

Finally, Dueling Double Deep Q-Learning (DDDQN) combines Dueling DQN with Double DQN to find the optimal Q-value, where the output from the Dueling DQN is passed to Double DQN. [61]

2.2 Simulation Environment

In this section an overview of components necessary is collected, followed by a brief description of software available for each component. Afterwards, some past projects that used these software are gathered.

2.2.1 Necessary components

In a first instance, the necessary requirements to develop and test the work proposed in this thesis were:

1. **Physics simulator and Rendering** - these are interlinked and will be used to visualize the correct operation of the proposed algorithm and attempt to get the most accurate representation of the real word;

2. **Network simulator** - to have accurate simulation of all communication between several UAVs and the GBS;
3. **Robotics framework** - to program each UAV's operating routine;
4. **Autopilot** - attached to the previous framework an autopilot will be required, being the bridge between robotics routines and control of the UAV model.
5. **AI framework** - to test the RL algorithms.

2.2.2 Available options

Amongst the several options the author came across in the literature, the preliminary research resulted in the options summarized in Table 2.1, for the Robotics framework, ROS clearly stood out as the preferred choice given its popularity. ROS2 was not considered since most of the relevant projects found were developed in ROS. Given the author's lack of experience with any, the safest decision to learn was ROS since more resources in the context of UAV exploration were found for this version.
The main criteria to choose which software to use were:

- being open-source;
- popularity, which meant more resources would be expected to be available and also maintained;
- scalability, given it was expected to test a multi-agent system;

Autopilot	Physics simulator and rendering	Network Simulators	AI framework
PX4 Autopilot	Gazeboo	ns-3	OpenAI gym
Ardupilot	V-REP (CoppeliaSim)	OMNeT+	RLlib
	AirSim		
	Unity Engine		

Table 2.1: List of software

Proceeding to describing and analysing each option presented in Table 2.1, by category, starting with ROS, it would be used to program each UAV, allied to an autopilot system. ROS is an open-source software framework for building robot applications. It provides a set of libraries and tools for developing distributed and decentralized systems, including multi-robot coordination and control. ROS includes a range of packages and plugins for UAVs, such as the MAVROS package (which will be addressed bellow) for communication with PX4-based UAVs.

2.2.2.A Autopilot

As for PX4 Autopilot, it is an open-source autopilot system for UAVs, that supports decentralized coordination and control. It provides a modular architecture that allows users to configure and customize the

behavior of multiple UAVs in a swarm. PX4 also includes support for a range of sensors and communication protocols, providing some models to use in ROS, thus making it well-suited for indoor exploration and mapping applications, such as the case of this work.

An alternative to PX4 Autopilot is ArduPilot, given that both systems have ROS packages that enable communication between the autopilot and ROS, allowing for the integration of UAVs with ROS-based applications and simulations.

2.2.2.B Physics simulator and rendering

Gazebo would be used to visualize in real time the scenarios that are being tested. According to the official website [29] , Gazebo is a accurate physics simulator, allowing for easier visualization. It has the advantage of having integration with ROS and given its popularity there is a lot of documentation and resources for developing, namely available scenarios and robot models, in particular UAV models. Gazebo's modular architecture includes a plugin system that permits users to extend and customize the simulator's functionality. Custom plugins can model specific sensors, control algorithms, or even entire robotic platforms, which means several sensors and noise models are already developed and maintained, specifically LiDAR sensors, and in the future they can be easily exchanged.

An alternative to Gazebo would be CoppeliaSim, also known as V-REP (Virtual Robot Experimentation Platform). It is a versatile and advanced robot simulation software. It is widely used for various purposes, including robotics algorithm development, simulation, and control. Users can design robots, program their behaviors, and simulate their interactions with the surrounding environment. It supports a wide range of sensors, actuators, and robotic platforms, making it suitable for simulating diverse robotic systems in 3D scenarios. CoppeliaSim's user-friendly interface and powerful simulation capabilities make it a popular choice in the field of robotics research and development, but given its simplicity and lack of documentation it was hard to integrate in the desired architecture.

Unity Engine is a powerful and versatile real-time 3D development platform widely used for creating interactive simulations, video games, Virtual Reality (VR), and Augmented Reality (AR) applications. The major setback of this option is the fact it does not have integration with ROS on its own, making it unfit for the desired architecture, despite offering robust graphics and extensive customization options, that enable the creation of visually appealing and interactive environments.

AirSim [30], short for "Air Simulation," is an open-source simulator that specifically targets the simulation of autonomous systems, including UAVs and other robotic platforms. It provides a realistic physics engine and sensor simulation, making it suitable for training and testing AI algorithms. It is compatible with various platforms, including ROS, PX4-Autopilot and Unreal Engine, and it enables the experimentation of state-of-the-art perception, planning, and control algorithms in a safe and controlled virtual environment. At the time, Gazebo was more explored than AirSim given its popularity and the fact that

AirSim was tested on Ubuntu 18.04 LTS, as mentioned in the official website [31]. This option was still mentioned since it is a complete and possible alternative.

2.2.2.C Network simulators

NS-3 [32] would be used to simulate the interactions between UAVs. NS-3 stands for Network Simulator Version 3, it is an open-source discrete-event network simulator. It allows the simulation of complex network scenarios and study their behavior under various conditions. On the other hand, OMNeT+ is also an open-source and based on discrete-event simulation but follows a component-based modeling approach, having a modular and more general approach. It also allows the study of the behavior and performance of UAV communication systems in various scenarios. OMNeT+ may be easier to use given it has Graphical User Interface (GUI) and a high-level simulation language called NED (Network Description Language). [41] In the final scope of this project a network simulator was not used, but usually researchers choose between these tools based on their specific requirements, familiarity with the platform, and the complexity of the simulation scenarios they aim to explore.

2.2.2.D AI framework

OpenAI Gym would be used to efficiently iterate and perfect the DRL algorithms developed. Is an open-source toolkit for developing and comparing RL algorithms. It provides a collection of environments, ranging from simple to complex, like Atari games, robotics simulations or automation of industrial processes. Additionally, it seamlessly integrates with popular RL libraries such as TensorFlow and PyTorch. The first official formal documentation about it can be found in [51] alongside the developing documentation for Gym [52]. A side note is that, Gym has been updated to Gymnasium in 2022 as announced in the official website [42]. OpenAI gym was chosen instead of Gymnasium given that more resources were available and several incompatibilities arose.
In addition, RLlib [100] is the industry-standard RL Python framework built on Ray. Designed for quick iteration and a fast path to production, it includes more than twenty five of the latest algorithms that are all implemented to run at scale and in multi-agent mode. It supports both TensorFlow and PyTorch. It can be swiftly integrated with OpenAI Gym's environments. This would be one of the next steps to implement in future developments.

The initial choice was to use Gazebo and ROS allied to PX4 Autopilot with OpenAI Gym, given their popularity and the fact that the author had no prior knowledge at that time in any of the softwares that she came across, so it was expected that more resources for learning would be available. To summarize in a simpler view, the base goal architecture would be similar to what is represented in Figure 2.5. According

to this base structure the author searched for existing projects given the lack of prior knowledge with the software, these would serve as a model or starting point.

It is important to note that, using this architecture would easily allow the transition between simulation and real-life tests, given that ROS and PX4 Autopilot are ready to do this transition just by getting the correct hardware.

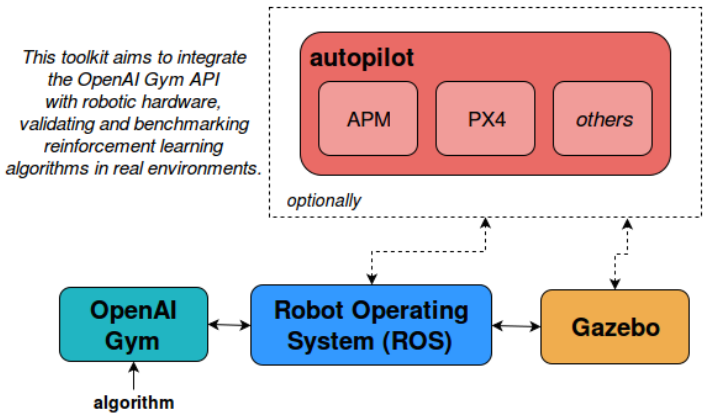


Figure 2.5: Simplified software architecture used in OpenAI Gym for robotics, from [4]

A short side note, MAVROS [53] would be used as the bridge between the autopilot and ROS. It is ROS package that provides communication drivers for various autopilots (compatible with PX4 and Ardupilot) with MAVLink communication protocol. Additionally, it provides UDP MAVLink bridge for ground control stations (e.g. QGroundControl [54], this was not necessary in this thesis, since a decentralized system was idealized and all the information in a simulated environment is accessible).

In the following subsection an overview of the main projects found that use the software mentioned in Table 2.1 and some further information are described.

2.2.3 Related projects

Starting with a project that involved RL, but did not match to the presented architecture is the work of Battocletti *et al.* [56], it presents a coordinated fleet approach where there are two different RL agents were developed: the first one coordinates the exploration task and delegates waypoints for each UAV in the fleet and the second agent is tasked with path planning and assigning the best routes for each UAV. It uses a Deep Deterministic Policy Gradient (DDPG) learning algorithm [57], which is a model-free[1], off-policy[2] learning algorithm suitable to work in continuous action space. It uses an actor-critic type

[1] model-free refer to a class of techniques where the agent learns to make decisions directly from interactions with the environment, without explicitly building an internal model of the environment's dynamics.

[2] off-policy refers to a class of algorithms where the agent learns a policy (a strategy for taking actions) that is different from the policy it uses to explore the environment and collect data.

of architecture, to concurrently learn a Q-value function and a policy. In this work a custom simulation environment has been implemented using Python 3.8.3, with some relevant libraries such as Tensorflow 2.3.0, Keras 2.4.3 and OpenCV 4.4.0. It does not have any integration with ROS or OpenAI Gym. The results obtained are for a 2D approach, but there is a simple implementation of the 3D environment available. The environment was traversed by a total of four UAVs with safe and dynamically-efficient trajectories, in environments rich in obstacles.

On the same note, a project that is ROS-based, but it is not an RL-based solution is RACER [58], it stands for RApid Collaborative ExploRation approach. It uses a fleet of decentralized UAVs for indoor exploration tasks and based on principles similar to frontier-based exploration. More precisely, an hierarchical planner was designed to find exploration paths and by refining local viewpoints, it can generate the shortest trajectories time-wise sequentially to explore the unknown space safely and agilely. The code developed was tested on ROS Kinetic and ROS Melodic, but not on ROS Noetic, which was the one used in this thesis. Several attempts were made to work with this project, but with no success.

During research, the idea of modeling the problem as a POMDP emerged, since it is commonly used in robotics navigation problems as previously mentioned. TAPIR [59] implements an algorithm developed by the authors: Adaptive Belief Tree (ABT) that reuses and improves existing techniques to quickly find a good approximate solution, and also introduces a novel capability to adapt the solution online in response to changes in the POMDP model. Furthermore, TAPIR provides an interface to the commonly-used ROS framework and CoppeliaSim (previously known as V-REP) simulator. The developed interface seemed promising and even fitted with using the POMDP to model the problem, but after testing CoppeliaSim, even though it has a simpler and user-friendly interface, it lacked the flexibility to create a custom application. For this reason, this approach was abandoned.

The work of Seel *et al.* [60] was the closest to what was intended in this thesis and was used as a main reference to build the developed environment, since the main goal was also to explore unknown GNSS-denied indoor environments, relying on IMU and LiDAR data. In [60], a LiDAR framework has been developed to enable a UAV to autonomously navigate and explore unknown factory environments, with the intent of mapping it. This framework is designed to meet specific requirements, such as operating in areas GNSS-denied areas and relying solely on onboard sensors. Data from IMU and LiDAR sensors is used for independent decision-making and to create a digital replica of the factory. The system has been implemented in ROS Gazebo, and various training and testing scenarios have been created for evaluation. Unfortunately, no source code was available for the ROS Gazebo models and testing environments that were developed. On another hand, a clear idea of which algorithm could be used was mentioned: DDDQN, which was identified as the best performing DRL algorithm regarding 3D UAV navigation by [63], being an improvement of previous versions, namely Dueling DQN [61] and Double Deep Q-Learning (DQN) [62], and Deep Q-Learning (DQN) itself as previously mentioned.

The following three approaches have integration with OpenAI Gym, but do not use PX4 Autopilot or Gazebo (in parenthesis is the direct link to the corresponding github repository and a brief description for each is found bellow):

- Gym-gazebo [64]
- Flightmare [70]
- Gym-pybullet-drones (the updated repository is [67], but at the time of development, the available repository was the one in the branch "paper")

Gym-gazebo [4] is an extension of OpenAI Gym for robotics using ROS and Gazebo, where a total of six environments for three different robots: Turtlebot, Erle-Rover and Erle-Copter were developed. Two RL algorithms were tested and compared using the developed tool kit, specifically Q-Learning and Sarsa [65]. Even though this repository was archived and the environments for UAVs (Erle-Copter models) were deprecated, the author still tried to use this in order to learn and build upon it, with no success given several compatibility issues once again. Moreover, this project did not include any autopilot yet, but had the desired integration between ROS, Gazebo, OpenAI Gym and had models for UAVs included.

A second version of this software called gym-gazebo2 [71] based on ROS2 was developed in the mean time but has also been archived, which can be found in [72]. Given that this work was based on ROS and not ROS2, it was not used.

Flightmare [69] is an open-source simulation platform designed for quadrotors. It is constituted by two main components: a configurable rendering engine built on Unity and a flexible physics engine for dynamics simulation. Flightmare is compatible with OpenAI Gym, having a flexible interface with stable baselines for solving tasks with DRL algorithms, but for a single agent RL workflow and for that reason it was not used. Additionally, Flightmare offers a ROS wrapper, allowing seamless interaction with widely used ROS packages.

Gym-pybullet-drones [66] is an open-source OpenAI Gym environment based on PyBullet [68] for multi-agent RL. A side note about PyBullet, it is an easy to use Python module for physics simulation, robotics and deep reinforcement learning based on the Bullet Physics engine. It was not mentioned before since this was the only project found that used it. One major disadvantage, in the context of this thesis, is that it does not support LiDAR sensors. Even though it supports multi-agent systems, it only supports vision-based RL interfaces, which for other projects this may be an advantage. In Figure 2.6 is a printscreen of a table found in its official github repository , which highlights the features that gym-pybullet-drones has compared to other solutions focused on RL or Crazyflie - Crazyflie is a open source flying development platform with small and lightweight quadrotors compatible with PX4. For more information check Craze's online shop [73].

For the context of this work, the fact that it supports multi-agent Gym-like API was an advantage, but it did not meet the remaining criteria. Additionally, the updated version[3] has a wrapper for ROS2 and it was tested on Ubuntu 22.04, which did not match this thesis experimental setup.

	`gym-pybullet-drones`	AirSim	Flightmare
Physics	PyBullet	FastPhysicsEngine/PhysX	*Ad hoc*/Gazebo
Rendering	PyBullet	Unreal Engine 4	Unity
Language	Python	C++/C#	C++/Python
RGB/Depth/Segm. views	**Yes**	**Yes**	**Yes**
Multi-agent control	**Yes**	**Yes**	**Yes**
ROS interface	ROS2/Python	ROS/C++	ROS/C++
Hardware-In-The-Loop	No	**Yes**	No
Fully steppable physics	**Yes**	No	**Yes**
Aerodynamic effects	Drag, downwash, ground	Drag	Drag
OpenAI `Gym` *interface*	**Yes**	No	**Yes**
RLlib `MultiAgentEnv` *interface*	**Yes**	No	No
PyMARL integration	*WIP*	No	No

Figure 2.6: Overview table from gym-pybullet-drones github repository

Proceeding to solutions with the desired architecture, using OpenAI Gym, PX4 and ROS, the following toolkits and frameworks were found (in parenthesis is the reference to the corresponding github repository and a brief description for each is found bellow):

- Gym_px4 [74]
- Gym-gazebo-px4 [75]
- MultiUAV-OpenAIGym [77]

Gym_px4 is an OpenAI Gym environment for PX4 Gazebo Software In-The-Loop (SITL) using MAVROS. This environment enabled the use of any gym RL library available at the time it was developed, such as baselines, stable-baselines or Keras-RL to train low-level quadcopter controllers. It was developed for ROS melodic and compatible with python 3.6 or 3.7, which are not compatible with the working settings of this thesis (ROS noetic and python 3.8), even so an attempt to use this environment was done, trying to make the necessary adjustments, but with no success and it was very time consuming.

Gym_gazebo_px4 corresponds to the updated version of the previously mentioned gym_gazebo which integrates PX4 Autopilot. Unfortunately, even though it has integration with PX4, it maintained the deprecated environments and the documentation was not updated, making it much harder to use this software as a working base to develop upon.

[3] [last accessed in 21st October] the repository received updates two weeks ago

MultiUAV-OpenAIGym [76] is a versatile environment for autonomous UAVs that has been designed to support various communication services in different application contexts, such as wireless mobile connectivity, edge computing, and data gathering. Developed within the OpenAI Gym framework, this simulation replicates real operational scenarios. The application context of Brunori [76] was to create a multi-agent system made up by a variable number of UAVs which are able to provide one or more (up to three) services to cluster(s) of users who request it. The developed environment features the creation different 2D or 3D environments, but it was so focused on multi-service framework that it did not suit the context of exploration.

After many months trying and testing, back and forth the mentioned softwares, the decision to simplify and create an environment from scratch was made. It was built upon OpenAI Gym given its simple structure, compatibility and possibility to scale to multi-agent systems.

An overview of existing OpenAI Gym or Gymnasium environments can be found in Table 2.2, having into account whether multi-agent training is available or not.

It is important to note, these are third-party environments in the official OpenAI Gym website [78] , the same list is also available in the official website for Gymnasium [79] , but with a note stating "There are a large number of third-party environments using various versions of Gym. Many of these can be adapted to work with gymnasium (see Compatibility with Gym), but are not guaranteed to be fully functional." and for that reason even though the lists in both websites are almost the same, using Gym instead of Gymnasium seemed to be the safest choice.

Name	Multi-agent	OpenAI Gym or Gymnasium
PyFlyt	Yes	Gymnasium
gym-pybullet-drones	No*	Gym
MarsExplorer	No	Gym

Table 2.2: Overview of OpenAI Gym and Gymnasium environments from the official websites

*this side note refers to the fact that even though gym-pybullet-drones allows simulations with several UAVs, it only supports the training of a single agent.

PyFlyt [87] is a toolkit for evaluating RL algorithms on different UAVs. Utilizing the Bullet physics engine, it provides adaptable rendering choices, time-discrete physics simulation and the capability to accommodate custom drones of diverse designs, including biplanes, quadcopters, rockets. This option was found at the end of the development of this thesis, thus it was not explored. However it was worth mentioning given its relevance to the topic.

The following projects were used as inspiration for the development of this thesis and to create IndoorExplorers environment, only gathering features of all these it was possible to create it.

[80] is a study that aims to connect advanced DRL techniques with the challenge of exploring and covering unknown terrains. To achieve this the MarsExplorer environment was created, designed specifically for exploring unknown areas. It is a OpenAI Gym environment that transforms the original robotics problem into a RL scenario that can be addressed by various readily available algorithms. Any learned strategy can be directly applied to a robotic platform without the need for a complex simulation model of the robot's dynamics, simplifying the learning and adaptation process. It is based on a grid like world, where a single agent equipped with a simulated LiDAR explores the unknown area. MarsExplorer was used as a foundation to develop this thesis, taking most of its features it was possible to create IndoorExplorers (the name was chosen accordingly to its inspiration). The features used from this environment will be further addressed in Chapter 3. Additionally, four different state-of-the-art RL algorithms at the time were trained on the MarsExplorer environment, namely Asynchronous Advantage Actor Critic (A3C) [81], Proximal Policy Optimization (PPO) [83], Rainbow [82] and Soft Actor-Critic (SAC) [84]. Their performance was compared to human-level performance. Given the performance of PPO, its performance was then compared to a frontier-based approach. Demonstrating that the policy based on PPO could efficiently adapt to unknown terrain while ensuring the coverage of areas that are costly to revisit, highlighting the effectiveness of RL-based approaches in exploration tasks.

Ma-gym [86] is a collection of multi-agent environments based on OpenAI Gym (even though it is not on the official websites), IndoorExplorers was also based on the Predator-Prey environment of this collection. Predator-prey involves a grid world, in which multiple predators attempt to capture randomly slow-moving prey. Each predator has a view mask corresponding to their cardinal direction, which means that a prey is caught if it is within the field of view of at least one predator. This concept is similar to having a LiDAR sensor and for this reason it was also used as one of the foundations of this thesis.

Some other important environments that were consulted and are worth mentioning are:

- multiagent-particle-envs [88] (the github repository can be found in [90]) is a multi-agent particle world with a continuous discrete action space and observation space, which has some basic simulated physics applied - it has a collection of environments with different types of interactions among particles (the original particle environment in which multiagent-particle-envs was based on is [89]) and was consulted to see how communications could be added to the project.
- Minigrid [91] (the github repository can be found in [92]) consists of a collection of discrete grid-world environments to conduct research on RL problems and served as inspiration to obtain a basic knowledge of grid-worlds.
- Miniworld [91] (the github repository can be found in [93]) is a simple 3D interior environment simulator for RL and robotics research, specially 3D navigation problems where office and home environments and mazes are available. This could be interesting to integrate in future works.

Finally, from the official documentation of both OpenAI's Gym and Gymnasium, PettingZoo [85] (the

official github repository can be found in [94]) is the major reference for multi-agent environments. It is a Python library developed by OpenAI themselves, that provides a collection of RL environments for multi-agent research and allows the creation of custom environments. Unlike traditional single-agent environments, PettingZoo focuses on scenarios involving multiple agents interacting in various ways. It offers a diverse range of games and simulations, enabling the study complex interactions, cooperation, and competition between agents. To create a custom environment, there is a main API that follows Agent Environment Cycle (AEC) modelling scheme, introduced in [85], in which agents sequentially see their observation and agents take actions, then rewards are emitted from the other agents, and the next agent to act is chosen. It is proven in the same document, that Partially Observable Stochastic Game (POSG) are equivalent to AEC games. And since POMDPs are a specific type of POSG, then it would a compatible framework for the exploration problem. Additionally, there is also a secondary parallel API for environments where all agents have simultaneous actions and observations. Having this in mind and the fact that it has integration with Ray's RLlib, it could be an interesting framework to explore in order to improve IndoorExplorers. This is information was added given its relevance to the topic and part of the contextualization to proceed to the actual developed environment.

2.3 Summary

To summarize, in this chapter a journey was made from the initial conception of this project to the decision to create a simulation environment from scratch. Analysing all the components that would be required in a global view of the project and the options available for them. The main reasons for the development of a custom environment were due to a series of incompatibilities and outdated solutions that led to the search for a simpler alternative and simpler solution. This solution will be based on a OpenAI Gym framework, given its popularity, simplicity, scalability and the fact it resources to accommodate the desired requirements.

3

Developed work

Contents

In this chapter, a description of the developed environment is provided, alongside the assumptions that were made, its features and limitations.

3.1 Assumptions

Before describing the environment, a set of assumptions were made in order to simplify the problem and are listed below:

- The area to be explored is considered to be a discrete space, divided into cells, whose dimensions are known (its height and width);
- Each agent's position is known at all times;
- Communication delay, including transmission delay, is negligible, but the communication system may have a limited range;
- Agents perform their actions sequentially, even though the choice of action is done previously. The order in which agents take their actions is randomized, so there is no priority amongst agents;
- The maps of agents in communication range are simultaneously and instantaneously merged and then a copy is saved for each agent;
- There are no collisions amongst agents - this is coded to be impossible;
- Each agent corresponds to one UAV flying at a fixed height.

3.2 Description of the developed environment

This environment can handle up to four agents simultaneously, which can be spawn randomly in the map or in pre-defined positions set in the "settings.py" file. In this file, it is possible to set everything related to the topology of the environment (such as the height and width, how many obstacles and with what size, etc.), configurations of the agents (such as the number of agents, the LiDAR range and communication range, etc.), values for rewards and some rendering options (for instance, it is possible to toggle on some auxiliary prompts or the rendering of each agent's map to aid in debugging and for easier visualization). In Appendix A.2 an example of this file is provided, with comments that explain each setting. The maps to be explored are randomly generated and have some parameters to adjust them, namely the "obstacles" setting that is the absolute number of obstacles to be placed randomly, "obstacle_size" that sets the range of values to set the shape of the obstacle (which means the first value must the lower than the second one), the "number_rows" and "number_cols" define a number of rows and columns of obstacles to be placed in the map. It is also possible to set "noise" which is applied to the positions generated by the "number_rows" and "number_cols", varying the positions of

the obstacles around pre-established positions. Adjusting these settings allows one to control the level of generalization desired to train the algorithm in question - the map generation functions were used unchanged from the MarsExplorer environment.

In the folder "multi_agents" is the created environment based on the previously mentioned single-agent MarsExplorer environment, in which each agent has a designated colour, agent 1 is blue, agent 2 is red, agent 3 is green and finally agent 4 is yellow. The LiDAR emulation was used unchanged from the MarsExplorer environment, but the rendering was entirely changed and is based on the ma-gym Predator-Prey environment (both were mentioned in Chapter 2).

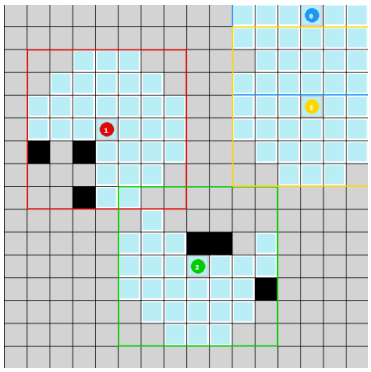
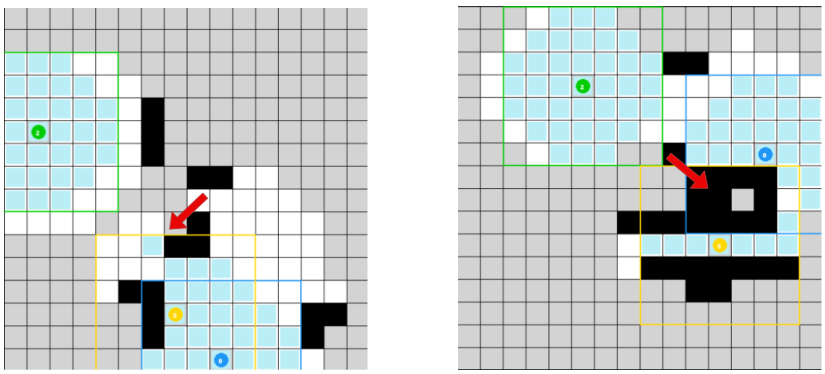


Figure 3.1: IndoorExplorers environment with 4 agents

As can be seen in Figure 3.1, the LiDAR field of view can be visualized by the light blue circle that surrounds each agent, the radius corresponds to the range defined in the settings file, in this case it is 3. There is a known behaviour about this emulation, which was disregarded since for a simple approach it was not compromising since a grid world is being considered. It consists in the fact that sometimes a ray of the LiDAR goes through an edge between two obstacle cells, enabling the agent to "see" a little bit further than it was supposed to - in images 3.2(a) and 3.2(b) it is possible to see this behaviour. In example 1, the agent in yellow can see a little bit further and in example 2, both agents in yellow and blue should not be able to see inside the obstacle nearby. This can be explained by the discretisation of the space in question.

Using OpenAI gym's simple configuration (more information can be found on the "core" section of the official website [52]) it is necessary to define four functions, which will be further described:

- step(action) - this function runs one timestep of the environment's dynamics, it is necessary to establish the core mechanisms for the environment. It receives an action that will be made effective inside this function and returns the information about the state, namely the observation, reward, terminated flag and some extra information.
- reset() - once the end of the episode is reached, this function must be called, which resets the environment to start a new episode. It only returns the initial observation.



(a) Example 1 **(b)** Example 2

Figure 3.2: Example of specific behaviour of the LiDAR emulation

- render() - function responsible for rendering.
- close() - usually closes the rendering window.

It is also necessary to have the algorithm to choose which actions to take, in this case it would be the DDDQN that will be addressed briefly. Additionally, the observation space and the action space must be defined, these specify whether the action space is discrete or continuous and what are their ranges. In this case, the MultiAgentObservationSpace and MultiAgentActionSpace from ma_gym were used. The observation space consists of a matrix that represents the map of what the agent sees, with float values ranging from 0,0 to the number of agents in the environment - each agent has a matrix called exploredMap which saves the updated real-time view of that agent (check Listing A.3 for the full description of the Agent class). In Table 3.1 are compiled the meaning of each value of the matrix corresponding to each agent's map.

Values	Meaning
0.0	Unexplored cell
0.3	Empty/explored cell
0.5	Obstacle
≥ 1	Id of corresponding agent occupying that cell

Table 3.1: Meaning of values in each agent's map

With the MultiAgentObservationSpace wrapper, the observation space is defined as a list with the maps of all the agents, with the indices matching each agent's id minus one, meaning that the observation with index zero corresponds to the map of agent number one.

As for the action space, for a single agent it takes one of four possible discrete actions depicted in Table 3.2, there is one additional action for the multi-agent case, that is no-op standing for no-operation to allow agents to not move in case it is more favourable. Using the MultiAgentActionSpace, the action

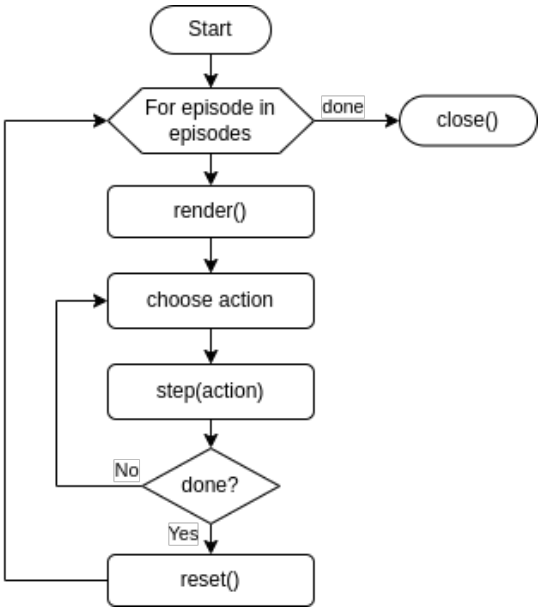


Figure 3.3: Flowchart of a basic OpenAI Gym application

space is the same but once again it becomes a list with the length of the number of agents, where each value is the action assigned to the corresponding agent.

Values	Corresponding action
0	Moving down
1	Moving left
2	Moving up
3	Moving right
4	No-op

Table 3.2: Meaning of each action

In Figure 3.3 and Appendix A.4, it is possible to see a flow chart that depicts the typical sequence of the previously described functions in a OpenAI gym environment for a single-agent scenario. Its simplicity is one of the reasons for its popularity. It consists of a loop with the desired number of episodes, that encloses a chain of choosing actions and then applying them in the step function. The step function then retrieves an *observation*, *reward*, the flag to whether the agent has entered a terminal state (*done*) and some extra information stored in *info*. When the done flag is activated the episode ends, triggering the reset of the environment, starting a new episode. The rendering is optional, but in this case it is done in the beginning of the loop. Once all episodes have been completed, the environment is closed.

In a multi-agent case, the structure is maintained, but each of the fundamental functions returns a list of variables, instead of a single reward the step function returns a list with the rewards for each agent, the same for the observations, and so forth. So, for example, when defining the multi-agent environment class, the step function would something similar to code found in the Listing 3.1 and the same for the

other functions.

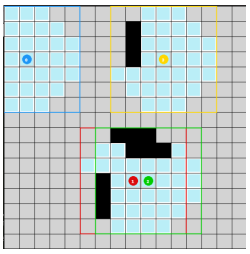
Listing 3.1: Example of step function for a multi-agent OpenAI gym class

```
class MultiAgentEnv(gym.Env):
    def step(self, action_n):
        obs_n    = []
        reward_n = []
        done_n   = []
        info_n   = {'n': []}
        # ...
        return obs_n, reward_n, done_n, info_n
```

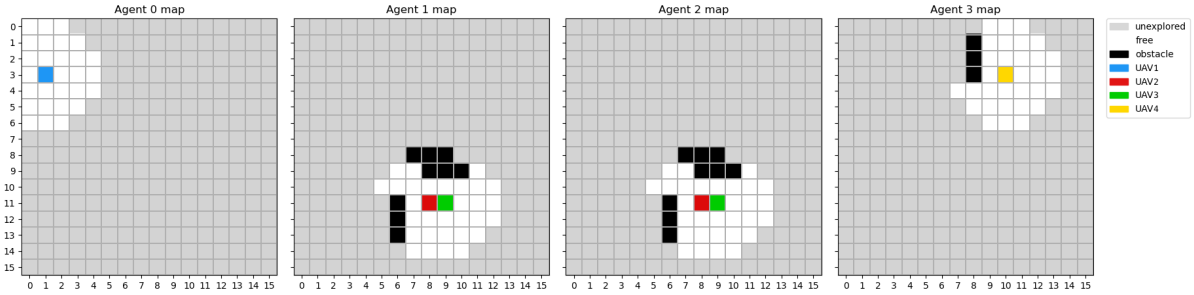
Since the structure for the multi-agent code is the same as the one in A.4, the actions for all agents are chosen first and stored in *action_n* array and this array is then used as an argument for the step function.

Inside the step function the order in which agents take action is randomized, thus avoiding any priority among agents. Before making the action effective, a verification is done to check if the move is valid, a collision with an obstacle has happened, if it takes the agent outside the maps' bounds, or if the agent gets stuck between positions. If the next move would result in a collision with another agent, then it is canceled and the agent does not move - this decision was made expecting it would result in faster learning. Regarding the getting stuck verification methods, this was implemented since a tendency for an agent to get stuck between positions was verified, this will be explained further in Chapter 4.

Additionally, inside the step function it is verified which agents are in communication range of others. In Figure 3.1 it is possible to see that the communication range of each agent is depicted by a square line surrounding the agent with its corresponding colour. The square is formed by a number of *c* cells to each direction of the center, where *c* corresponds to the defined communication range in the "setting.py" file, in this case it is 3.0. This verification is carried out by checking the connected components in an indirect graph where each node is an agent, using the auxiliary adjacency matrix *comm_range* (further mentioned in Table 3.3 and the Depth First Search (DFS) algorithm (the idea for this comes from [97]). Having the connected components, each agent's explored map (corresponds to the updated map of what it sees, also found bellow in Table 3.3) that is in range is then merged and a copy is saved in each, replacing the old explored map. In Figures 3.5 and 3.4 it is possible to observe the moment when all agents are in range of each other.

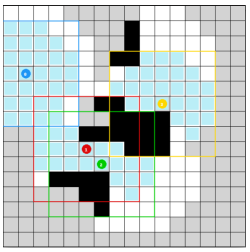


(a) Full view

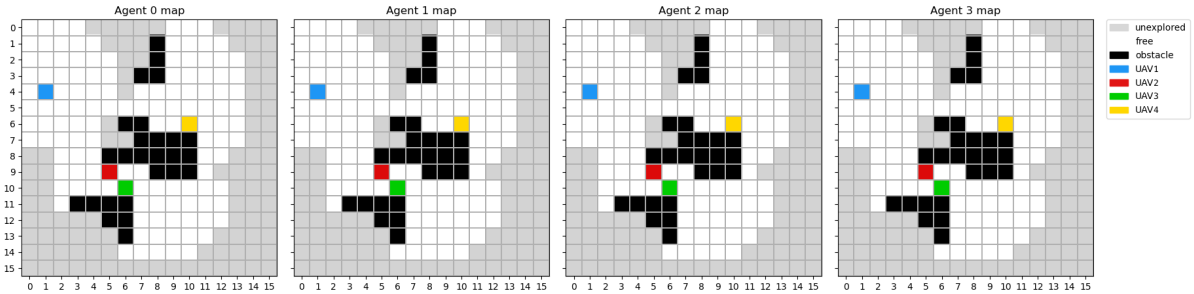


(b) Each agent's explored map

Figure 3.4: Merging of maps demo - before merging



(a) Full view



(b) Each agent's explored map

Figure 3.5: Merging of maps demo - after merging

After merging the maps, if possible, the reward is calculated, and a verification on whether each agent has reached a terminal state is done. The episode only ends when all agents have reached a terminal state - this includes one of the following options:

- reaching the maximum number of steps (defined as 400, in the settings file);
- a collision with a wall or obstacle has happened;
- the agent goes outside the bounds of the map;
- a cyclic path has been found or in simpler terms the condition to be considered stuck has been reached;
- the map has been explored by the percentage set in the settings file.

The reward system is explained in Chapter 4, since several values where experimented. The general idea is that positive outcomes, such as successfully exploring the defined percentage of the map (in the example in A.2, the value is 90%), the agent is rewarded with a large positive value, whereas in the other cases (except the reaching the maximum number of steps) which correspond to harmful or disadvantageous situations, the agent is penalized with a large negative reward. Regarding each new cell that is explored, a pre-defined value is added for each - a copy of the previously explored map is saved and then compared with the current map, by subtracting the number of non-zero cells, it is possible to get the number of newly explored cells.

An overview of each global matrix that was created can be found in Table 3.3, considering that the area to explore has height n and width m, the number of agents is *a* and *i* is an arbitrary index for one agent. These variables alongside the Agent class gather all the necessary information to know the state of every part of the environment.

Matrix	Dimensions	Purpose
groundTruthMap	n x m	Map with all the cells discovered
lidar_map	n x m	Simplified map used in lidar scans
_full_obs	n x m	Map with information gathered from all agents
agents[i].exploredMap	n x m	Each agent has a map with the information it has gathered
comm_range	a x a	Adjacency matrix, that represents which agents are in communication range of other agents

Table 3.3: Overview of the global matrices created

Before advancing into the RL algorithm that chooses which action the agents take, it is worth mentioning that one feature related to rendering is the possibility to visualize each agent's exploredMap by setting the "printMap" option to true in "settings.py", in Figure 3.5(b) for example, it is possible to see this feature.

3.3 Integration of RL algorithm

Advancing onto the algorithm, as stated the choice was to use DDDQN based on the work of [60]. In order to add this algorithm to the environment, an implementation from Robbie Estes was used as a foundation (the official github repository can be found in [98]) , who trained the network to play Joust, Ms. Pac-Man , Super Mario and Space Invaders, using gym retro [95] (the official github repository can be found in [96] - which takes classic video game roms into OpenAI Gym environments for RL training of fully capable player agents and comes with integration for over a thousand games, from different consoles such as Nintendo, Atari, NEC and Sega.

The architecture of a DDDQN can be seen in Figure 3.6 (this image is based on the image in [5]). The network was prepared to have as input a stack of image frames of the game being played and outputting a vector of Q-values for each action possible in the given state, taking the highest Q-value of this vector will give the best action for that state.

At the time, the dimensions of the network were adapted to receive a matrix which corresponds to the observation of the agent being trained. Initially, the matrix was converted to an image represented by three matrices for the RGB values of the corresponding image and these were stacked, but the agent was not learning as intended since the input space was to sparse. At the present time, the realization that a misunderstanding happened during this step, since this work was not a video game, then there was no need to use pre-processing of the video game frames and a few steps were removed - the map was already represented by a single matrix with well defined values. The belief that one crucial step was removed, namely, the stacking of frames which gives the network a sense of movement. This means the results were obtained using a single map, which can be considered as a single frame instead of a stack of frames - this will be further commented in Chapter 4.

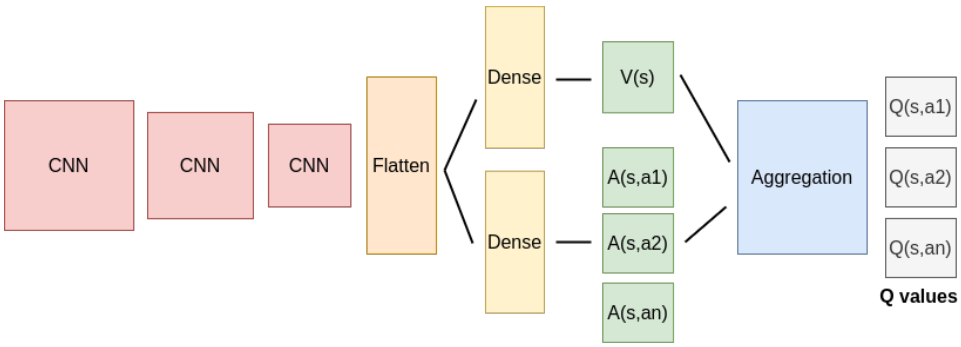


Figure 3.6: Architecture of DDDQN (image based on image from [5])

Moving on to a brief description of the network, the first three layers are used to process the input frames to extract features. After flattening, the information stream is divided into two components. To explain this, a reminder that Q-values, Q(s,a), correspond to how good it is to be in a given state *s* and

taking an action *a* at that given state. This means Q(s,a) can be decomposed as the sum of V(s) - the value of being at that state *s* - and A(s,a) - the advantage of taking action *a* at that state *s* (how much better is to take this action versus all other possible actions at that state) - as can be seen in equation 3.1.

$$Q(s,a) = A(s,a) + V(s) \tag{3.1}$$

Concerning the aggregation layer, to generate the Q values for each action in that state, it is necessary to subtract the average advantage of all actions possible of the state, as evidenced in equation 3.2, in order to avoid the issue of identifiability in back propagation - not being able to identify A(s,a) and V(s), given a certain Q(s,a).

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{A}\sum_{a'} A(s,a') \tag{3.2}$$

Finally as explained before DDDQN takes advantage of the features of the Double DQN in order to avoid overestimation of Q values. The accuracy of Q values depend on what actions have been tried and what neighboring states have been explored. As a consequence, at the beginning of the training, there is not enough information about the best action to take. Therefore, taking the maximum Q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, then the learning will be unstable. The solution is to use two networks to decouple the action selection from the target Q value generation. This is how the Double DQN helps to reduce the overestimation of Q values and, as a consequence, helps training faster and have more stable learning. The weights from the "q_eval" network (which is the name of the model in the code) are copied to the "q_target" model after *update_every* number of episodes.

3.4 Features

To summarize, at present time, the main features of this environment are stated as follows:

- It can perform simulations with up to four agents;
- Each agent has a circular field of view with a pre-defined range, constrained by an emulated LiDAR, this means the data is acquired as if there were laser scans, and each agent cannot "see" through walls/obstacles - the LiDAR emulation was extracted from [80] and left unchanged;
- There is communication between agents, which allows exchange of information - merging of their maps. There is an emulated communication range, which is delimited by a square with the agent in its centers and that extends in cardinal directions by the pre-defined range.

3.5 Limitations

There are three limitations to this environment and will be exposed in the following bullet points:

- The behaviour of the LiDAR emulation should be something to be considered and improved in the future;
- the misunderstanding in implementing the DDDQN did not allow to test the environment to its full capabilities - by improving the observation space and tuning the network for instance, the algorithm can be tested to its full capabilities;
- the limited implementation of other RL algorithms, such as stable-baselines3 [99]and other algorithms available in Ray RLlib [100], which would serve as benchmark - given the incompatibilities and struggles with package versions within the conda virtual environment, this was not implemented given the limited time to develop this project.

3.6 Summary

To summarize, in this chapter an overview of the developed environment is presented, alongside its features and limitations, the core algorithm used is explained. There is still space for improvements but in the next chapter, it is possible to see it is a robust environment to develop RL algorithms in the context of indoor exploration problems.

4

Results and Analysis

Contents

In this chapter, a compilation of the results obtained are presented. The goal was to test the performance of the developed environment, finding the best parameters for learning, even though the hyperparameters of the neural network were not tuned. Nonetheless, some other important aspects were tested:

1. influence of stuck verification methods, explained further in this chapter;
2. influence of reward values;
3. influence of number of agents
4. influence of communication

4.1 Experimental setup

All the simulations were executed on the author's personal computer, with a Intel® Core™ i7-8750H CPU @ 2.20GHz $\times$ 12 processor and NVIDIA Corporation GP107M [GeForce GTX 1050 Ti Mobile] / Mesa Intel® UHD Graphics 630 (CFL GT2) graphics card. In A.1 Listing is a list of the requirements that were used in a conda virtual environment.

As for the DDDQN's hyperparameters, the values used are from the original source code for the DDDQN [98] and are presented in Table 4.1.

Hyperparameter	Value	Purpose
batch_size	32	batch size
learn_every	10	interval of steps to fit model
update_every	10.000	interval of steps to update target model
alpha	0,0001	learning rate
gamma	0,99	discount factor
epsilon	1,0	exploration factor
epsilon_min	0,01	minimum exploration probability
epsilon_decay	0,99999	exponential decay rate of epsilon
memory_size	100.000	replay memory size

Table 4.1: DDDQN's hyperparameters

4.2 Results

It was verified that for a single agent in a 16x16 dimension map with no obstacles and a maximum numbers of 400 steps, it could get stuck between two positions and for that reason two simple methods to verify if it was stuck were implemented:

- **no stuck verification (no stuck)** - there is no verification and it is shortly designated as "no stuck" in the following plots.
- **stuck verification method 1 (stuck 1)** - In this method, if the agent has not discovered any new cell for *height*width* of the map steps, then it is stuck. The reason for the *height*width* value is that with 100% certainty any agent can transverse the whole map in that amount of steps, even thought this value is the worst case scenario, since it means the agent passes through each cell one time. For abbreviation, this method will be referred as "stuck 1".
- **stuck verification method 2 (stuck 2)** - For this method, an history of the past fifty positions of the agent is saved, if the most frequent one is repeated at least twenty times, then it is considered that the agent is stuck. This method will also be referred as "stuck 2".

Three scenarios with different values for rewards were tested, summarized in Table 4.2. The values for scenario 1 are based on the work [80] and the values for scenario number 2 are based from [60]. Finally, scenario 3 was created from the previous with minor adjustments.

- **Scenario 1** has the exact same values as [80], having the movement cost or the cost per step be half of the reward gained when discovering a new cell, correspondingly -0,5 and +1. Additionally, all the negative actions (collision with obstacles, going outside the bounds of the map and getting stuck) have the same value of -400 and the bonus reward, which is assigned when exploring a given percentage of the map (in this case 90%), with a value of +400.
- **Scenario 2** uses the exact values of negative rewards for collisions with obstacles and getting out of the bounds of the map as [60], precisely -100.000, but a different value for getting stuck, being -10.000. The aim of having different values is such that collisions and getting out of the bounds have greater consequences in a real scenario than being stuck. The values for movement cost and reward for discovering a new cell are also the same as in [60], being -1 and +10 correspondingly.
- **Scenario 3** seeks to test the weight of rewarding a bonus as done in [80], adding the assumption that every negative action has the same value, and the movement cost has the symmetric value of the reward for exploring a new cell. Basically, all have the same value but negative actions have a negative value, while positive actions have positive value - therefore, movement cost has a value of -10, while the reward per new cell discovered is +10. For every negative action the value is -1.000 and the bonus reward is +1.000. For the value of each movement cost, the value -10 was chosen to check if the agent would choose shorter routes, making each step more valuable, having a proportion of one to one, instead of half as done in scenario 1.

In order to see which reward scenario would have a stabler and robust learning, each combination of the three scenarios with the three possible stuck methods were tested in a 16x16 map with no obstacles

Name	Value			Purpose of the reward
	scenario 1	scenario 2	scenario 3	
Movement cost	-0,5	-1	-10	Value that is discounted per step
Collision	-400	-100000	-1000	Negative reward for colliding with an obstacle
Out of bounds	-400	-100000	-1000	Negative reward for getting outside of the bound of the map
Stuck	-400	-10000	-1000	Negative reward when it is stuck between positions
New cell discovered	+1	+10	+10	Positive reward for discovering a new cell
Bonus reward	+400	+10000	+1000	Positive reward when the given percentage of the map is explored (in this case 90%)

Table 4.2: List of rewards for different scenarios

for 10.000 episodes. For every trained model, an evaluation was done after a pre determined number of episodes, in this case every 200 or 500 episodes, and a record of it was saved, these results will be commented alongside the presented graphs, even though it is not possible to present these videos in this document. It is important to note that the moments of intermediary evaluation do not affect training, they are only checkpoints to collect data.

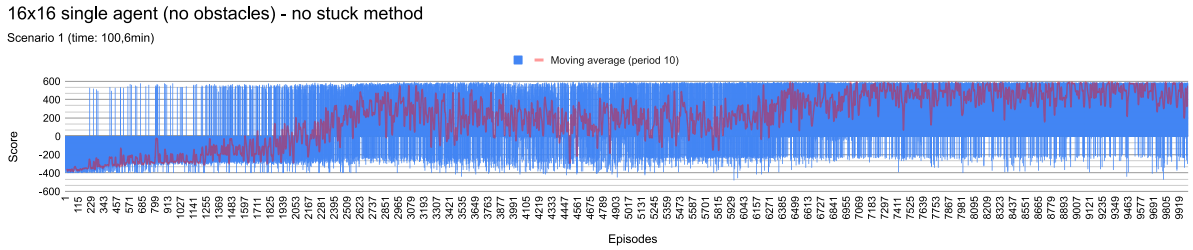
A small note, in Appendix B it is possible to find zoomed out versions of some figures, where the labels are not readable but the plots are easily discerned - precisely all figures related to the percentage of the map explored in relation to the number of steps.

In Figure 4.1, it is possible to observe the score per episode for scenario 1, the score corresponds to the sum of rewards during the entire episode. In each figure it is possible to see how long the training lasted and in all the cases the time was directly linked with the number of times the agent got stuck, thus a longer training means the agent got stuck more often.

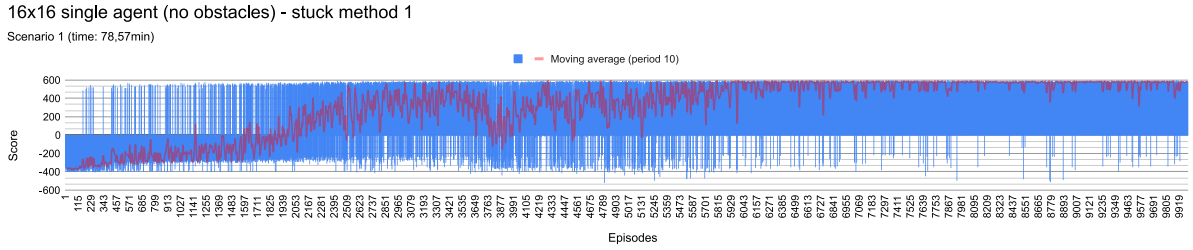
Having this in mind, it can be verified that negative score values correspond to situations where a negative outcome happened, but in scenario 1 it is not possible to verify which of the negative outcomes occurred in each episode. On the other hand, every value of score above the bonus rewards value (400) is associated to having explored 90% of the map.

By looking at the three graphs and the moving average with a period of 10, it seems that stuck method 1 produces less negative outcomes and produces a stabler value regarding the score, while stuck method 2 produces the most negative outcomes. But when looking at the percentage of the area explored with respect to the number of episodes in each step, in Figure 4.2, it is possible to confirm that stuck method 2 allows a faster and more complete exploration of the map.

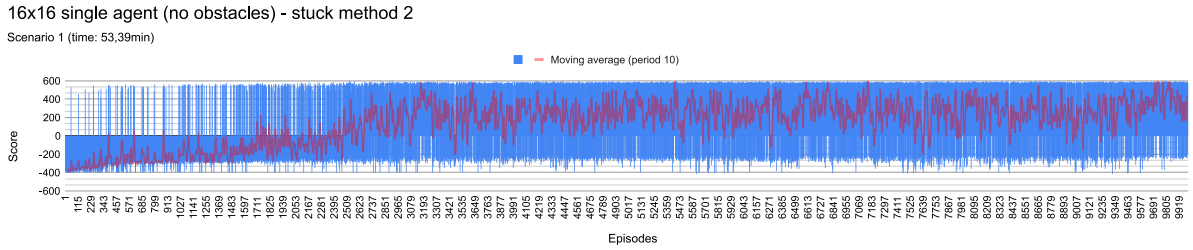
Regarding the recordings of each stuck method, in the last evaluations, the agent learned how to transverse the map in a circular clockwise or counter-clockwise motion which is the most efficient way, as illustrated in Figure 4.3, with stuck methods 1 and 2. While with no stuck method the agent persists to get stuck between positions.



(a) No stuck method



(b) Stuck method 1



(c) Stuck method 2

Figure 4.1: Scores per episode in scenario 1 with no obstacles

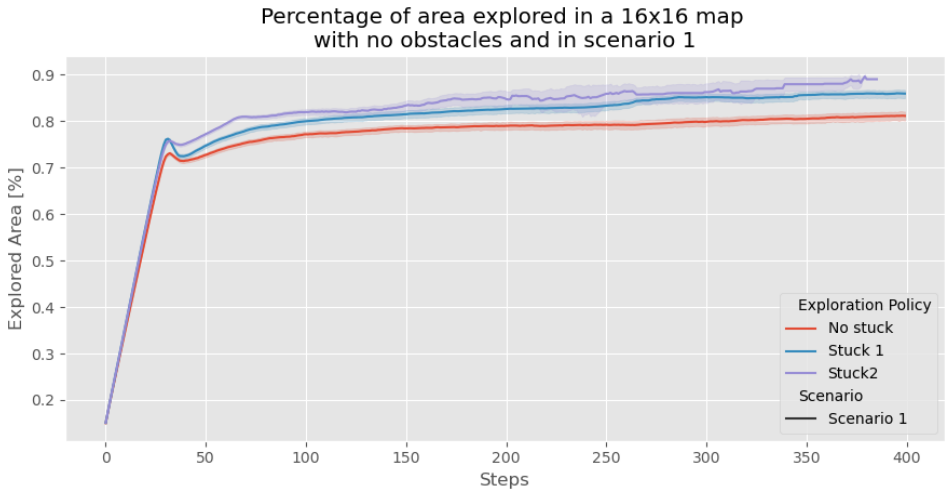


Figure 4.2: Percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles in scenario 1

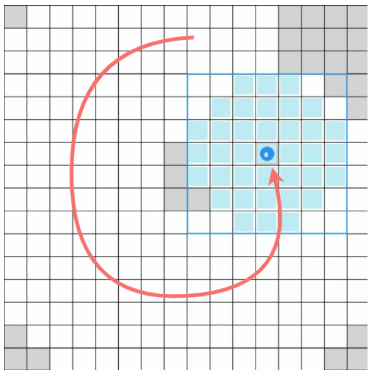
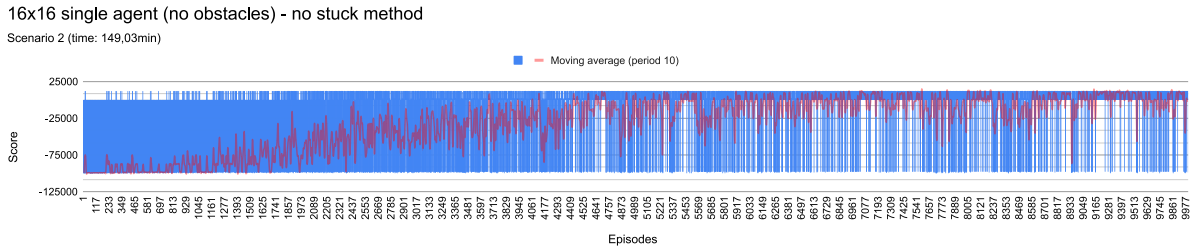
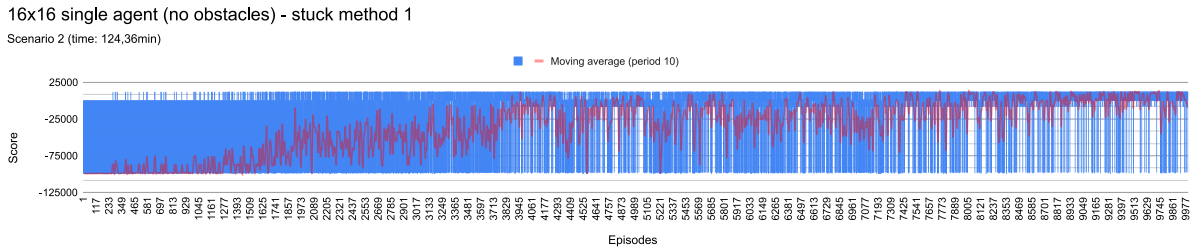


Figure 4.3: Illustration of the optimal trajectory learnt in a 16x16 area with no obstacles in scenario 1 with stuck method 1 and 2

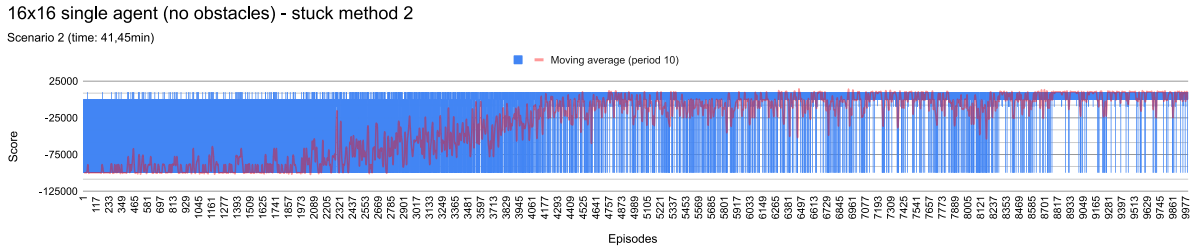
Proceeding to scenario 2, in Figure 4.4, it can be observed that the moving average is very similar for all three stuck methods. Looking at the reward values in this scenario, it is possible do discriminate the moments in which collisions with obstacles occur or the moments where the agents get out of bounds from the cases where the agent gets stuck. It appears that in stuck method 2 there are less identifiable moments where it gets stuck.



(a) No stuck method



(b) Stuck method 1



(c) Stuck method 2

Figure 4.4: Scores per episode in scenario 2 with no obstacles

By watching the recordings, the same behaviour is observed, the agent also learns the circular motion in order to explore the map in any stuck method, nevertheless with no stuck method the agent still gets stuck. Moreover, in evaluations in the middle of the training, it was observed that the agent explored longer trajectories, which makes sense given the ration between reward for each new cell explored (+10) and the negative reward attributed to each step (-1).

It is noticeable in Figure 4.5 that using no stuck method or using stuck method 1 have a stabler exploring process, however with stuck method 2 the number of steps never reaches the maximum value of 400, and in some episodes achieves similar results in less steps than the other two approaches.

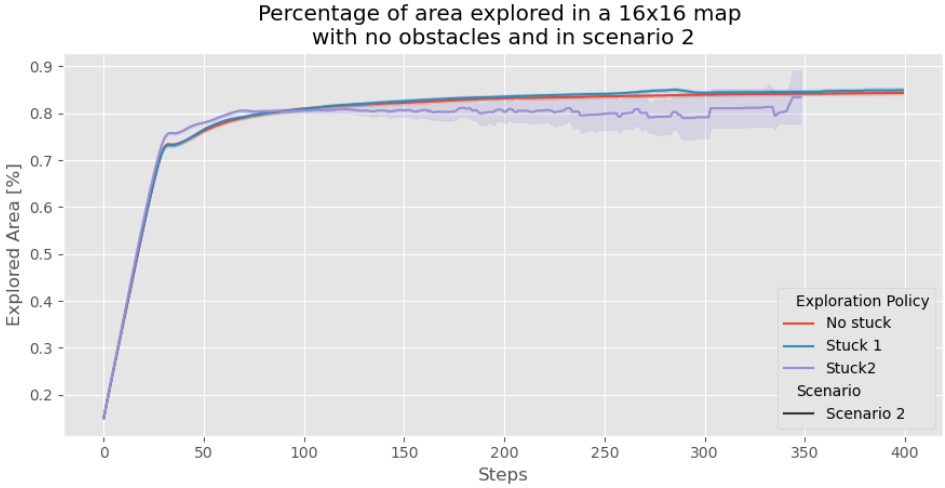
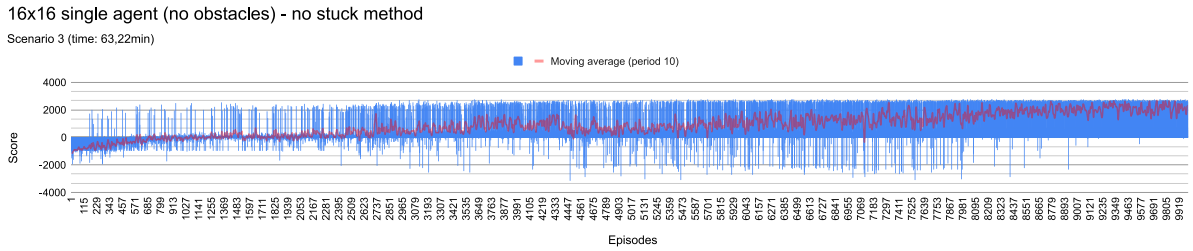
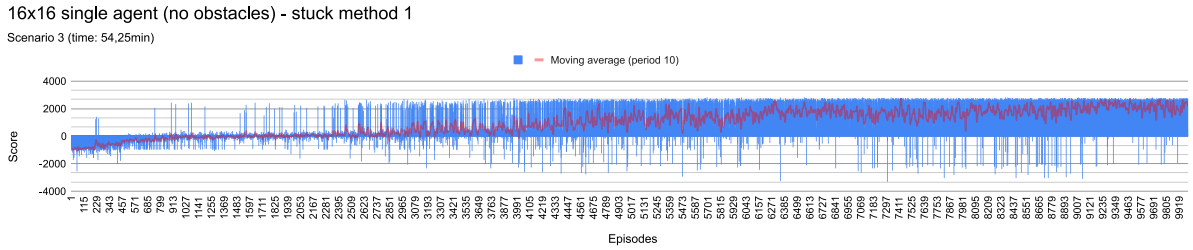


Figure 4.5: Percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles in scenario 2

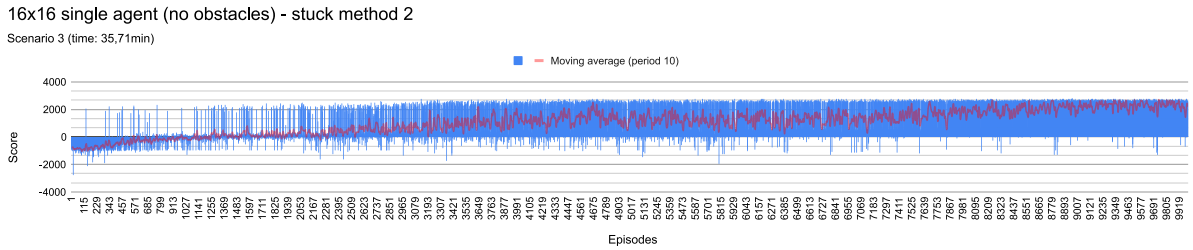
Proceeding to scenario 3, in Figure 4.6, stuck method 2 seems to have less negative outcomes while learning earlier how to avoid them - by looking at the moving average which crosses the zero score value earlier (between the 1.020th and the 2.000th episode).



(a) No stuck method



(b) Stuck method 1



(c) Stuck method 2

Figure 4.6: Scores per episode in scenario 3 with no obstacles

With regard to the recordings, once again the behaviour is similar to the previous scenarios, in every case, the circular pattern is learnt, although with no stuck method the agent still gets stuck and between stuck method 2 and method 1, the first one gets stuck less times. Observing Figure 4.7, it is evident that stuck method 1 seems to have a stabler curve, while stuck method 2 is able to explore more area with less steps - most of the episodes with this method result in exploring the map in less that 275 episodes in the successful cases which are the most frequent ones.

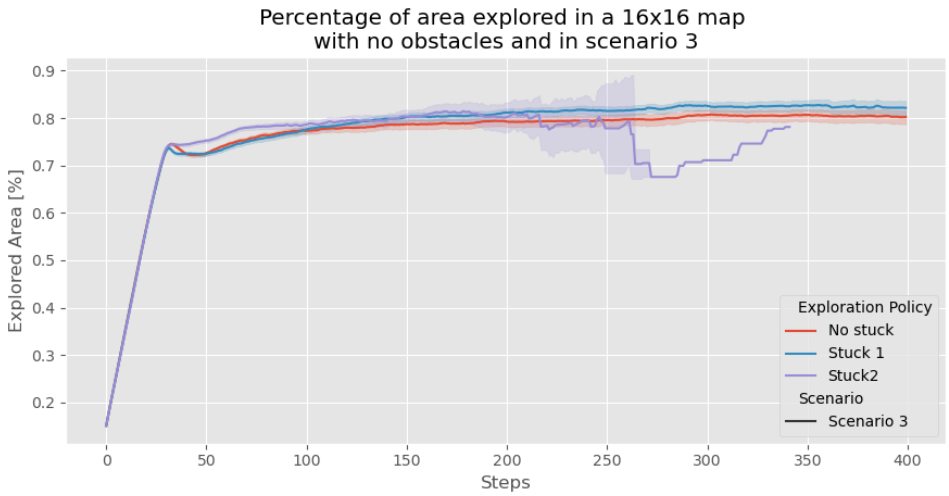


Figure 4.7: Percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles in scenario 3

As can be seen in Figure 4.8, the method that provides the best coverage in a fewer number of steps is stuck method 2 with scenario 1.

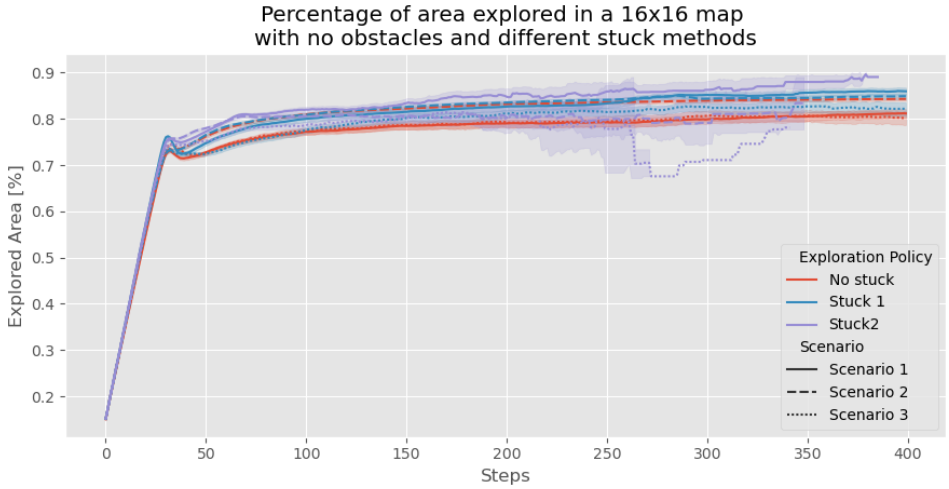


Figure 4.8: Overview of percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles

Having these three scenarios into account and the results obtained with no obstacles, the combination of stuck method 2 with scenario 1 will be used for the next tests, in which five obstacles of size 3x3 are now inserted randomly in the map (this will be labeled as "obstacles 5" in some figures for short).

Firstly, a single agent was trained for a longer number of episodes, concretely 50.000 episodes, all of the following tests were trained with this value. In Figure 4.9, it possible to observe that most of the episodes result in a negative score which is associated with the augmented number of collisions, as expected. By observing the moving average, it is visible that it has a small positive slope, which indicates a tendency for the score to increase and less collisions to be occurring.

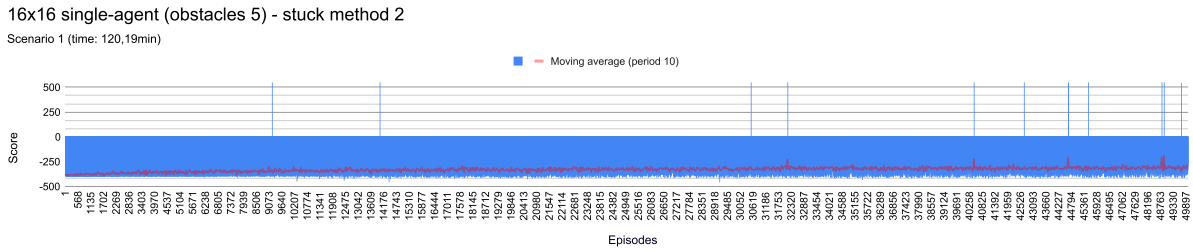


Figure 4.9: Score per episode for a single agent in scenario 1 and stuck method 2 with 5 obstacles

Looking at the recordings, it is possible to see in the last evaluations that the agent learns how to avoid some obstacles, such as seen in Figure 4.10 - this evaluation was done on episode 48.000 and in its next step the agent collides with an obstacle.

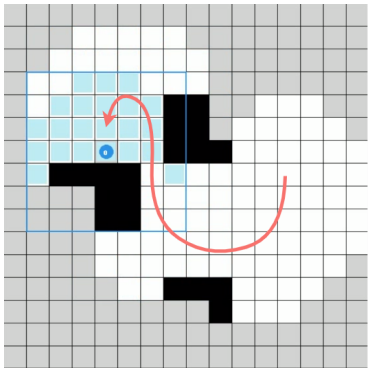


Figure 4.10: Illustration of a trajectory learnt in a 16x16 area with 5 obstacles in scenario 1 with stuck 2

In all following tests, the model trained with 50.000 iteration for a single-agent with stuck method 2 is loaded, and the training of new models is done using only agent 1 (the blue one), while the remaining agents choose random actions. In evaluation moments, the learnt model until that time is used by all the other agents to choose their actions. The following models are trained with two agents in the map and maintaining the stuck method 2, but in different communication scenarios.

Starting with the case where there are no communication, which corresponds to having the communication range equal to zero. In Figure 4.11, it is possible to verify that in this case, with two non-communicating agents, there is an increase in the number of episodes with a positive score. Observing the recordings, it is noticeable that if any of the agents do not get stuck, both seem to try to take the same path. In Figure 4.12, it is possible to see an example of this example, where they take similar paths, but there are other cases where the path is identical.

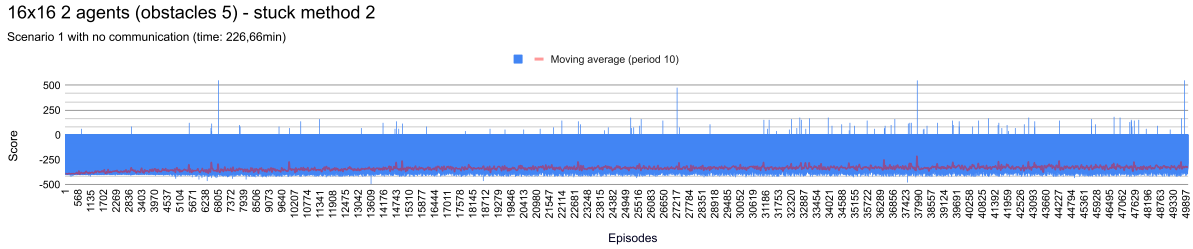
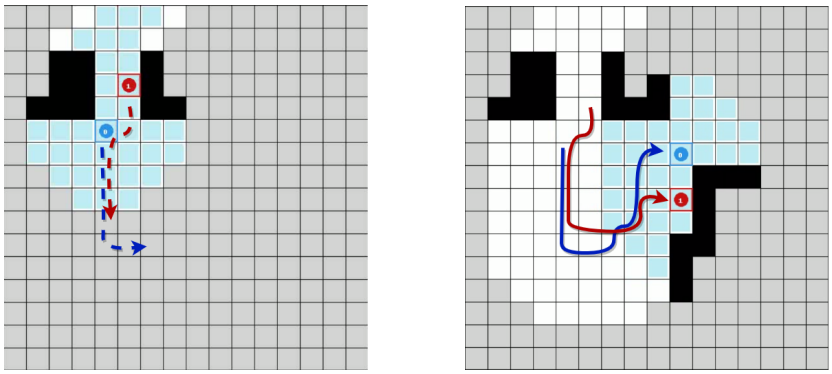


Figure 4.11: Score per episode with two agents and rewards from scenario 1, with stuck method 2 and no communication



(a) Initial steps

(b) One step before agent 2 collides with an obstacle

Figure 4.12: Example of the trajectories by 2 agents with no communication

The next tests with two agents and the previously defined settings, introduce communication with a range of 3.0 and 1.0 cells. Comparing the case with communication range 3.0 or just "comms 3" for short with the case where the range is 1.0 ("comms 1"), the shorter range has a higher incidence of positive scores, visible in Figures 4.14 and 4.13, respectively. The moving average in all the cases with two agents has a very similar curve. Looking at the recordings, a similar behaviour to what was found previously happens, if any of the agents do not get stuck, both seem to try to take the same path, as can be seen in Figure 4.15 where the two agents do not enter each others communication range. When agents enter each other's communication range, they synchronize and start taking the same actions - an explanation will be provided in the next section.

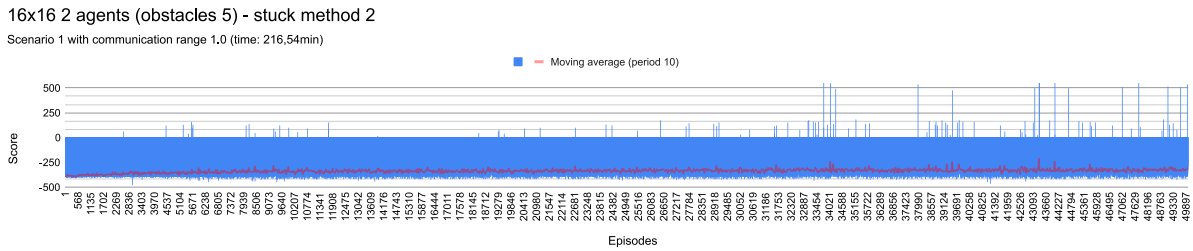


Figure 4.13: Score per episode with two agents, in scenario 1, with stuck method 2 and communication range 1.0

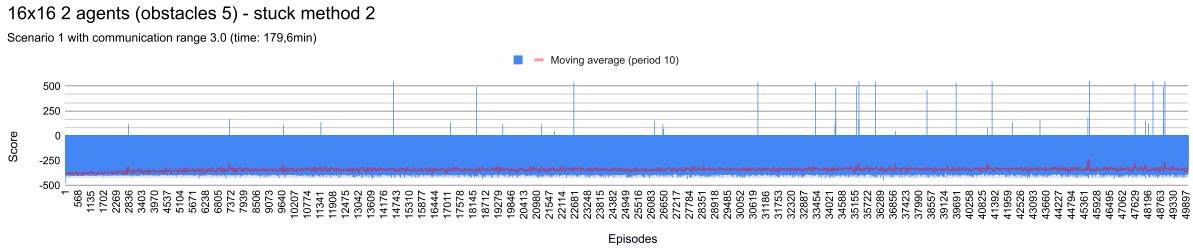
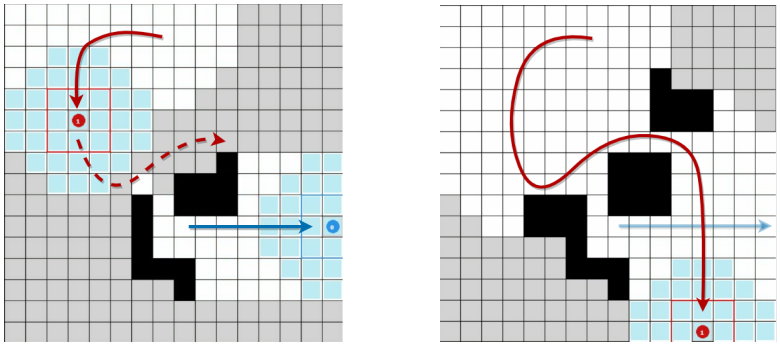


Figure 4.14: Score per episode with two agents, in scenario 1, with stuck method 2 and communication range 3.0



(a) Initial steps - before agent 2 gets out of bounds

(b) Last steps

Figure 4.15: Example of the trajectories by 2 agents with communication range 1.0

In Figure 4.16, it can be observed that the approach with no communication equalises to the approach with communication range 3.0 with fewer steps. This was expected since one of the goals of having communication is to reduce the distance each agent as to traverse and avoid visiting areas previously visited by other agents.

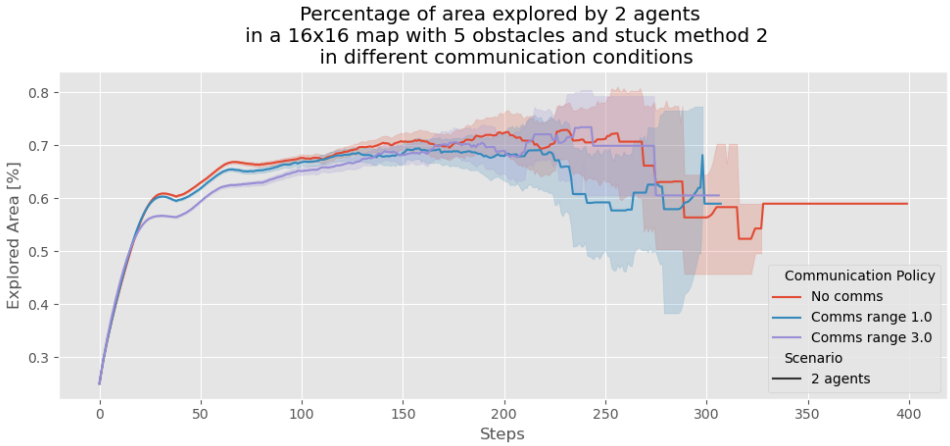
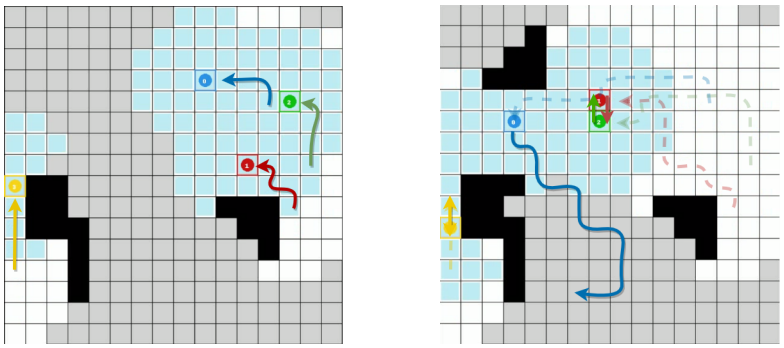


Figure 4.16: Percentage of the area explored with respect to the number of steps, by 2 agents various communication ranges in a 16x16 area with 5 obstacles

The final test that was done has four agents with no communication, in the same setting as the previous tests: loading the model learnt by a single agent in a map with five obstacles and only one agent

trains the model (agent 1 - the blue one), in evaluation moments the trained model is used by each agent. The choice to not use communications was due to the fact the agents synchronized movements and got stuck when in communication range. In Figure 4.18, it can be observed that relatively to the other previous multi-agent tests, the increased number of agents led to the increase of positive score values. This was also expected since the higher the number of agents, the higher is the area each can cover. By observing the recordings, it is possible to visualize that even on the last evaluations moments, there are several agents that get stuck between positions or choose actions that move them towards other agents, getting them stuck in the same position repeatedly and generally only one agent manages to explore the map, as can be seen in Figure 4.17 - where agent 3 (yellow) gets stuck between two positions, agent 2 (green) and agent 1 (red) choose repeatedly to move in each other's direction - since no collisions between agents are allowed, they get stuck - and only agent 1 (blue) does a valuable trajectory.



(a) Initial steps **(b)** Last steps

Figure 4.17: Example of the trajectories by 4 agents with no communication - evaluation in episode 48.500

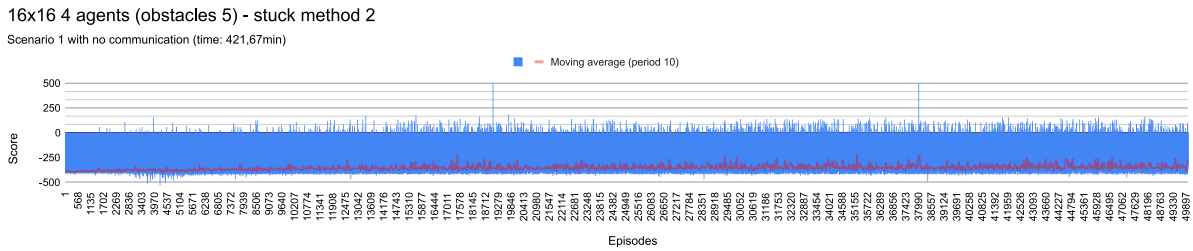


Figure 4.18: Score per episode with four agents and rewards from scenario 1, with stuck method 2 and no communications

Given the unexpected behaviour of several agents synchronizing, the comparison of all the multi-agent cases with no communications regarding the percentage of the map that is explored per step is compared in 4.19. As expected, the higher the number of agents, higher is the area covered in less steps. In the multi-agent cases, the stuck scenarios are associated with the highest number of steps and that is why the values of the percentage of the area explored decrease and even stagnate with

higher number of steps.

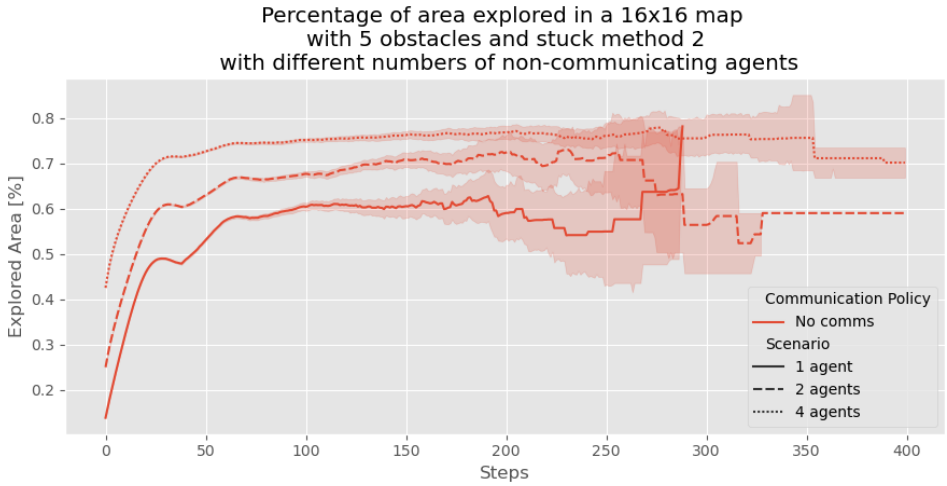


Figure 4.19: Percentage of the area explored with respect to the number of steps, by various number agents and no communication ranges in a 16x16 area with 5 obstacles

Finally, in 4.20 all the multi-agents approaches are compared and it is evident that the four-agent approach surpassed every other approach. This result was expected, for the reasons mentioned before. However, the use of communications were expected to have a better performance - this will be further analysed in the next section.

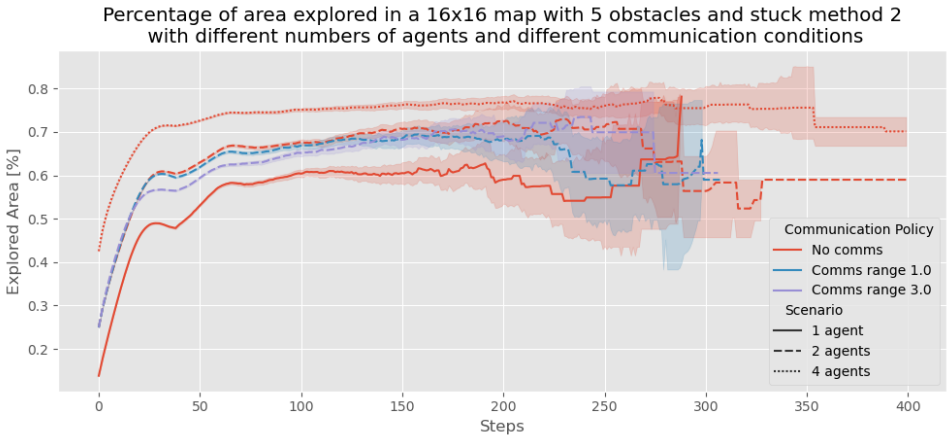


Figure 4.20: Overview of the percentage of the area explored with respect to the number of steps, in a 16x16 area with 5 obstacles

4.3 Discussion

The first expected result that was verified was the fact that having more agents reduces the number of steps it takes to explore a given space. By opposition, an unexpected behaviour was observed when communications were introduced. A possible explanation can be due to the fact that the agent only

receives the explored map as an observation and this has a direct impact in the exploration processing, concretely in the moments where all agents synchronize their decisions when in communication range, since what they see is exactly the same. In order to improve this behaviour, changing the observation space should provide faster learning and a more accurate behaviour, for example using a tuple with the agent's id, its position, their explored map and information about the other agents' positions - a similar approach was used in the ma-gym's Predator-Prey environment. With these changes, each agent should be able to identity themselves and differentiate itself from other agents when they merge their maps, thus avoiding taking the same action as the other agents. Another improvement could be providing a list of frontier points and known obstacles instead of the whole map, in an attempt that the observation space would not be so sparse, hence improving and accelerating the learning process.

Another unexpected behaviour that was seen was the fact that the agents would get stuck between positions. To explain this and as referred in the previous chapter, there is the belief that one crucial step in the implementation of the DDDQN is missing, namely the stack of frames which provides the notion of motion and past movements. The behaviour where the agents get stuck between positions seems to be deeply related to this missing component. Unfortunately, since this error was detected in at a belated stage of this thesis, it was not possible to present tests and results, but it is a common practice in the implementation of the DDDQN whose inclusion can be noticed by the presented results.

Additionally, the fact that there are no collisions allowed between agents adds another undesired behaviour, it was intended to accelerate the training process but instead it promotes the decision to go against other agents without consequence, meaning that they get stuck choosing the same action of moving in the direction of another agent repeatedly and being counter productive.

Another interesting observation, was the fact that with different ratios of reward for each new cell explored and penalisation for each step taken, the agents seemed to add value to each step taken. For instance, during training and looking at the recordings, when the reward for each new cell explored was 10 and the step cost was -10, the agent learnt in an earlier stage to take shorter routes, whereas when the reward was 10 and the penalty was -1, the agent would take longer and more intricate paths. And the middle term was using the second scenario's reward system, where the reward was +1 and -0.5 was the penalty for each step taken.

Despite these behaviours, it is possible to see that the DDDQN allows a single-agent to learn an optimal path, such as the case where there are no obstacles and the agent learns a circular pattern, in clockwise or counterclockwise motion. With the corrections mentioned in this section and the proper tuning of the neural network, DDDQN shows promising results and the developed environment seems to be simple and robust to develop and test more algorithms.

5

Conclusion

Contents

In this final chapter, a summary of the achieved goals and conclusions that this study allowed to gather, alongside the limitations and improvements that can be done are presented.

5.1 Conclusions

In this work, it was successfully developed a functional multi-agent OpenAI Gym environment for indoor exploration in GNSS-denied environments, in which up to four agents can be simulated with a simple emulated communication system where exploration information can be shared, concretely their maps.

It was possible to verify the impact that the lack of frame stacking in the DDDQN implementation has, precisely, it removes the sense of direction of each agent, thus presenting unexpected behaviour such as having agents stuck between positions. Before getting to this conclusion, simple methods to detect whether agents are stuck were implemented, which overall did not solve the main issue.

Additionally, the impact of several agents was tested and as expected more agents is in general a better solution. Unfortunately the impact of communication could not be properly tested. The results showed that the agents could not identify themselves and when in reach of each other, they synchronized their actions, not being able to search the area properly.

5.2 System Limitations and Future Work

Having reached the end of this project, it is far from being a mature solution, but it is a solid starting point with many aspects to improve, such as:

- Resolving the misunderstanding in the implementation of the DDDQN, which did not allow to test the environment to its full capabilities and it can be fixed with the proper adjustments mentioned in Chapter 4;
- properly tuning the DDDQN, it should greatly improve performance and accelerate the training process;
- it could also be interesting the addition of the 3D component, to which the DDDQN is proven to have good results, as stated in Chapter 2;
- possibly improve the LiDAR emulation, this should produce more accurate results;
- the implementation of other RL algorithms, these should be added to serve as benchmark, such as stable-baselines [99] and other algorithms available in Ray's RLlib [100].

Bibliography

[1] S. Chen, W. Zhou, A.-S. Yang, H. Chen, B. Li, and C.-Y. Wen, "An end-to-end uav simulation platform for visual slam and navigation," *Aerospace*, vol. 9, no. 2, 2022. [Online]. Available: https://www.mdpi.com/2226-4310/9/2/48

[2] G. Fevgas, T. Lagkas, V. Argyriou, and P. Sarigiannidis, "Coverage path planning methods focusing on energy efficient and cooperative strategies for unmanned aerial vehicles," *Sensors*, vol. 22, p. 1235, 2 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/3/1235

[3] F. AlMahamid, S. Member, and K. Grolinger, "Reinforcement learning algorithms: An overview and classification." [Online]. Available: https://www.ieee.org/publications/rights/copyright-

[4] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, "Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo," 8 2016. [Online]. Available: http://arxiv.org/abs/1608.05742

[5] T. Simonini, "Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed. . .," image source. [Online]. Available: https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/

[6] TU Delft/MAVLab, "The drones explore the office environment to find disaster 'victims'," 2019, source of the cover image [accessed Feb 16, 2023]. [Online]. Available: https://www.imeche.org/news/news-article/swarm-of-tiny-drones-finds-%27disaster-victims%27-with-minimal-computing-power

[7] N. Boonyathanmig, S. Gongmanee, P. Kayunyeam, P. Wutticho, and S. Prongnuch, "Design and implementation of mini-uav for indoor surveillance." IEEE, 3 2021, pp. 305–308. [Online]. Available: https://ieeexplore.ieee.org/document/9440350/

[8] I. Mademlis, V. Mygdalis, N. Nikolaidis, M. Montagnuolo, F. Negro, A. Messina, and I. Pitas, "High-level multiple-uav cinematography tools for covering outdoor events," *IEEE Transactions on Broadcasting*, vol. 65, pp. 627–635, 9 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8630599/

[9] J. Dong, K. Ota, and M. Dong, “Uav-based real-time survivor detection system in post-disaster search and rescue operations,” *IEEE Journal on Miniaturization for Air and Space Systems*, vol. 2, pp. 209–219, 12 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9440534/

[10] H. A. Foudeh, P. C.-K. Luk, and J. F. Whidborne, “An advanced unmanned aerial vehicle (uav) approach via learning-based control for overhead power line monitoring: A comprehensive review,” *IEEE Access*, vol. 9, pp. 130 410–130 433, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9528303/

[11] D. Rakesh, N. A. Kumar, M. Sivaguru, K. V. R. Keerthivaasan, B. R. Janaki, and R. Raffik, “Role of uavs in innovating agriculture with future applications: A review.” IEEE, 10 2021, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9675612/

[12] A. Tahir, J. Böling, M.-H. Haghbayan, H. T. Toivonen, and J. Plosila, “Swarms of unmanned aerial vehicles — a survey,” *Journal of Industrial Information Integration*, vol. 16, p. 100106, 12 2019.

[13] A. Chriki, H. Touati, H. Snoussi, and F. Kamoun, “Fanet: Communication, mobility models and security issues,” *Computer Networks*, vol. 163, p. 106877, 11 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128618309034

[14] G. Beni and J. Wang, “Swarm intelligence in cellular robotic systems,” pp. 703–712, 1993, the first time SI was mentioned/defined. [Online]. Available: http://link.springer.com/10.1007/978-3-642-58069-7_38

[15] S. A. H. Mohsan, M. A. Khan, F. Noor, I. Ullah, and M. H. Alsharif, “Towards the unmanned aerial vehicles (uavs): A comprehensive review,” *Drones*, vol. 6, p. 147, 6 2022. [Online]. Available: https://www.mdpi.com/2504-446X/6/6/147

[16] M. Campion, P. Ranganathan, and S. Faruque, “Uav swarm communication and control architectures: a review,” *Journal of Unmanned Vehicle Systems*, vol. 7, pp. 93–106, 6 2019. [Online]. Available: http://www.nrcresearchpress.com/doi/10.1139/juvs-2018-0009

[17] B. Yamauchi, “A frontier-based approach for autonomous exploration.” IEEE Comput. Soc. Press, 1997, pp. 146–151. [Online]. Available: http://ieeexplore.ieee.org/document/613851/

[18] S. Karaman and E. Frazzoli, “Sampling-based algorithms for optimal motion planning,” vol. 30, 6 2011, pp. 846–894, in this paper it has been presented the results of a thorough analysis of sampling-based algorithms for optimal path planning.

[19] J. D. Gammell and M. P. Strub, “Asymptotically optimal sampling-based motion planning methods,” *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 295–318, 5 2021. [Online]. Available: https://www.annualreviews.org/doi/10.1146/annurev-control-061920-093753

[20] M. A. Hassan, G. Kulathunga, and A. Klimchik, “Exploration and mapping of an indoor environment using multirotor aerial vehicle.” IEEE, 8 2021, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/9666129/

[21] K. Mcguire, “Indoor swarm exploration with pocket drones,” 11 2019. [Online]. Available: https://doi.org/10.4233/uuid:48ed7edc-934e-4dfc-b35c-fe04d55caee1

[22] M. Juliá, A. Gil, and O. Reinoso, “A comparison of path planning strategies for autonomous exploration and mapping of unknown environments,” *Autonomous Robots*, vol. 33, pp. 427–444, 11 2012. [Online]. Available: http://link.springer.com/10.1007/s10514-012-9298-8

[23] A. Xu, C. Viriyasuthee, and I. Rekleitis, “Optimal complete terrain coverage using an unmanned aerial vehicle.”

[24] U. S. Technology, “Drone lidar airborne lidar systems for uav.” [Online]. Available: https://www.unmannedsystemstechnology.com/expo/drone-lidar/

[25] G. Torres, “Photogrammetry vs. lidar: what sensor to choose for a given application.” [Online]. Available: https://wingtra.com/drone-photogrammetry-vs-lidar/

[26] Mathworks, “What is slam?” [Online]. Available: https://www.mathworks.com/discovery/slam.htmlhttps://www.mathworks.com/discovery/kalman-filter.htmlhttps://www.mathworks.com/help/nav/ug/perform-lidar-slam-using-3d-lidar-point-clouds.html

[27] B. Gerkey, “gmapping ros package.” [Online]. Available: https://wiki.ros.org/gmapping

[28] ——, “slam_karto ros package.” [Online]. Available: https://wiki.ros.org/slam_karto

[29] Gazebo, “Gazebo official website.” [Online]. Available: https://gazebosim.org/home

[30] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “Airsim: High-fidelity visual and physical simulation for autonomous vehicles,” in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[31] AirSim, “Airsim official website.” [Online]. Available: https://microsoft.github.io/AirSim/build_linux/

[32] NS-3, “Ns-3 official website.” [Online]. Available: https://www.nsnam.org/

[33] N. Perumal, T. Kalaiselvi, P. Nagaraja, Z. A. Basith, R. Scholar, and M. P. Scholar, “A comprehensive study on glowworm swarm optimization,” 2017, has a table comparing 4 SI (Swarm Intelligence) algorithms. [Online]. Available: https://www.researchgate.net/publication/313401910

[34] L. M. Pyke and C. R. Stark, “Dynamic pathfinding for a swarm intelligence based uav control model using particle swarm optimisation,” *Frontiers in Applied Mathematics and Statistics*, vol. 7, 11 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fams.2021.744955/full

[35] A. Tam, “A gentle introduction to particle swarm optimization,” 9 2021. [Online]. Available: https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/

[36] J. TORRES.AI, “Deep reinforcement learning explained,” pp. 1–2, 7 2020. [Online]. Available: https://torres.ai/deep-reinforcement-learning-explained-series/

[37] C. Nicholson, “A beginner’s guide to deep reinforcement learning.” [Online]. Available: https://wiki.pathmind.com/deep-reinforcement-learning#three

[38] Y. Wang, P. Wang, J. Zhang, Z. Cui, X. Cai, W. Zhang, and J. Chen, “A novel bat algorithm with multiple strategies coupling for numerical optimization,” *Mathematics*, vol. 7, p. 135, 2 2019. [Online]. Available: https://www.mdpi.com/2227-7390/7/2/135

[39] M. Popović, T. Vidal-Calleja, G. Hitz, J. J. Chung, I. Sa, R. Siegwart, and J. Nieto, “An informative path planning framework for uav-based terrain monitoring,” *Autonomous Robots*, vol. 44, pp. 889–911, 7 2020. [Online]. Available: http://link.springer.com/10.1007/s10514-020-09903-2

[40] X. Zhou, F. Gao, X. Fang, and Z. Lan, “Improved bat algorithm for uav path planning in three-dimensional space,” *IEEE Access*, vol. 9, pp. 20 100–20 116, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9334996/

[41] J. Weiss, “How do you compare ns-3 and omnet++ for wireless ad hoc network simulations?” 2023. [Online]. Available: https://www.linkedin.com/advice/0/how-do-you-compare-ns-3-omnet-wireless

[42] F. Foundation, “Announcing the farama foundation,” 10 2022. [Online]. Available: https://farama.org/Announcing-The-Farama-Foundation

[43] C.-C. Peng and R. He, “A concept for high precision digital terrain 3d mapping using uavs and multiple solid state lidars,” in *2022 IEEE International Conference on Consumer Electronics - Taiwan*, July 2022, pp. 457–458.

[44] M. Aljehani, M. Inoue, and T. Yokemura, “Particle swarm optimization algorithm presented in sysml and applied in multi-uav system,” vol. 2021-January. IEEE, 1 2021, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/9427618/

[45] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf, “A flexible and scalable slam system with full 3d motion estimation,” in *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011. [Online]. Available: https://wiki.ros.org/hector_slam

[46] S. Macenski and I. Jambrecic, “Slam toolbox: Slam for the dynamic world,” *Journal of Open Source Software*, vol. 6, p. 2783, 5 2021. [Online]. Available: https://wiki.ros.org/slam_toolbox

[47] W. Hess, D. Kohler, H. Rapp, and D. Andor, “Real-time loop closure in 2d lidar slam.” [Online]. Available: https://wiki.ros.org/cartographer

[48] M. Lauri, D. Hsu, and J. Pajarinen, “Partially observable markov decision processes in robotics: A survey,” 9 2022. [Online]. Available: http://arxiv.org/abs/2209.10342http://dx.doi.org/10.1109/TRO.2022.3200138

[49] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, 2 2015.

[50] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning.”

[51] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 6 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[52] Gym, “Gym documentation.” [Online]. Available: https://www.gymlibrary.dev/

[53] V. Ermakov, “Mavro ros package.” [Online]. Available: https://wiki.ros.org/mavros

[54] QGroundControl, “Qgroundcontrol simulator.” [Online]. Available: http://qgroundcontrol.com/

[55] M. Calvo-Fullana, D. Mox, A. Pyattaev, J. Fink, V. Kumar, and A. Ribeiro, “Ros-netsim: A framework for the integration of robotic and network simulators,” 1 2021. [Online]. Available: http://arxiv.org/abs/2101.10113

[56] G. Battocletti, R. Urban, S. Godio, and G. Guglieri, “Rl-based path planning for autonomous aerial vehicles in unknown environments.” American Institute of Aeronautics and Astronautics, 8 2021. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2021-3016

[57] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” 9 2015. [Online]. Available: http://arxiv.org/abs/1509.02971

[58] B. Zhou, H. Xu, and S. Shen, “Racer: Rapid collaborative exploration with a decentralized multi-uav system,” 9 2022. [Online]. Available: http://arxiv.org/abs/2209.08533

[59] D. Klimenko, J. Song, and H. Kurniawati, “Tapir: A software toolkit for approximating and adapting pomdp solutions online.” [Online]. Available: http://robotics.itee.uq.

[60] A. Seel, F. Kreutzjans, B. Kuster, M. Stonis, and L. Overmeyer, “Deep reinforcement learning based uav for indoor navigation and exploration in unknown environments.” IEEE, 4 2022, pp. 388–393. [Online]. Available: https://ieeexplore.ieee.org/document/9782602/

[61] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling network architectures for deep reinforcement learning,” 11 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[62] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 9 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[63] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, “Automatic drone navigation in realistic 3d landscapes using deep reinforcement learning,” in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, April 2019, pp. 1072–1077.

[64] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, “gym-gazebo github repository.” [Online]. Available: https://github.com/erlerobot/gym-gazebo

[65] G. Rummery and M. Niranjan, “On-line q-learning using connectionist systems,” *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[66] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, “Learning to fly – a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control,” 3 2021. [Online]. Available: http://arxiv.org/abs/2103.02142

[67] ——, “gym-pybullet-drones github repository.”

[68] Bullet, “Bullet real-time physics simulation.” [Online]. Available: https://pybullet.org/wordpress/

[69] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, “Flightmare: A flexible quadrotor simulator,” in *Conference on Robot Learning*, 2020.

[70] Song, Yunlong, Naji, Selim, , Kaufmann, Elia, Loquercio, Antonio, Scaramuzza, and Davide, “Flightmare github repository.” [Online]. Available: https://github.com/uzh-rpg/flightmare

[71] N. G. Lopez, Y. L. E. Nuin, E. B. Moral, L. U. S. Juan, A. S. Rueda, V. M. Vilches, and R. Kojcev, “gym-gazebo2, a toolkit for reinforcement learning using ros 2 and gazebo,” *https://github.com/AcutronicRobotics/gym-gazebo2*, 3 2019.

[72] ——, “gym-gazebo2 github repository.” [Online]. Available: https://github.com/AcutronicRobotics/gym-gazebo2

[73] Bitcraze, “Crazyflie 2.1.” [Online]. Available: https://store.bitcraze.io/products/crazyflie-2-1

[74] “gym_px4 github repository.” [Online]. Available: https://github.com/Benykoz/gym_px4

[75] Zamora, Iker, Lopez, N. Gonzalez, Vilches, V. Mayoral, Cordero, and A. Hernandez, “gym-gazebo-px4 github repository.” [Online]. Available: https://github.com/HHM98/gym-gazebo-px4/tree/master

[76] D. Brunori, S. Colonnese, F. Cuomo, and L. Iocchi, “A reinforcement learning environment for multi-service uav-enabled wireless systems,” 5 2021. [Online]. Available: http://arxiv.org/abs/2105.05094http://dx.doi.org/10.1109/PerComWorkshops51409.2021.9431048

[77] ——, “Multiuav-openaigym github repository,” *https://github.com/DamianoBrunori/MultiUAV-OpenAIGym*. [Online]. Available: https://github.com/DamianoBrunori/MultiUAV-OpenAIGym

[78] “Third-party environments on openai gym’s official website.” [Online]. Available: https://www.gymlibrary.dev/environments/third_party_environments/

[79] “Third-party environments on gymnasium’s official website.” [Online]. Available: https://gymnasium.farama.org/environments/third_party_environments/

[80] D. I. Koutras, A. C. Kapoutsis, A. A. Amanatiadis, and E. B. Kosmatopoulos, “Marsexplorer: Exploration of unknown terrains via deep reinforcement learning and procedurally generated environments,” *Electronics*, vol. 10, p. 2751, 11 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/22/2751

[81] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” 2 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[82] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” 10 2017. [Online]. Available: http://arxiv.org/abs/1710.02298

[83] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 7 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[84] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” 1 2018. [Online]. Available: http://arxiv.org/abs/1801.01290

[85] J. K. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sullivan, L. Santos, R. Perez, C. Horsch, C. Dieffendahl, N. L. Williams, Y. Lokesh, and P. Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning," 9 2020. [Online]. Available: http://arxiv.org/abs/2009.14471

[86] A. Koul, "ma-gym: Collection of multi-agent environments based on openai gym." https://github.com/koulanurag/ma-gym, 2019.

[87] J. J. Tai, J. Wong, M. Innocente, N. Horri, J. Brusey, and S. K. Phang, "Pyflyt – uav simulation environments for reinforcement learning research," 4 2023, http://arxiv.org/abs/2304.01305; https://pypi.org/project/PyFlyt/.

[88] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," 6 2017. [Online]. Available: http://arxiv.org/abs/1706.02275

[89] I. Mordatch and P. Abbeel, "Emergence of grounded compositional language in multi-agent populations," *arXiv preprint arXiv:1703.04908*, 2017.

[90] R. Low, Y. Wu, A. Tamar, A. Tamar, P. Abbeel, and I. Mordatch, "multiagent-particle-envs github repository." [Online]. Available: https://github.com/openai/multiagent-particle-envs

[91] M. Chevalier-Boisvert, B. Dai, M. Towers, R. de Lazcano, L. Willems, S. Lahlou, S. Pal, P. S. Castro, and J. Terry, "Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks," *CoRR*, vol. abs/2306.13831, 2023, [Online] Available: https://github.com/Farama-Foundation/Minigrid; https://github.com/Farama-Foundation/Miniworld.

[92] ——, "Minigrid github repository." [Online]. Available: https://github.com/Farama-Foundation/Minigrid

[93] ——, "Miniworld github repository," *https://github.com/Farama-Foundation/Miniworld*. [Online]. Available: https://github.com/Farama-Foundation/Miniworld

[94] "Pettingzoo github repository."

[95] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in rl," *arXiv preprint arXiv:1804.03720*, 2018.

[96] Nichol, Alex, Pfau, Vicki, Hesse, Christopher, Klimov, Oleg, Schulman, and John, "Openai retro github repository." [Online]. Available: https://github.com/openai/retro

[97] GeeksforGeeks, "Connected components in an undirected graph." [Online]. Available: https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/

[98] R. Estes, “Github repository with implementation of dddqn.” [Online]. Available: https://github.com/rjalnev/DDDQN/tree/master

[99] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” pp. 1–8, 2021. [Online]. Available: https://github.com/DLR-RM/stable-baselines3.

[100] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, “Rllib: Abstractions for distributed reinforcement learning,” 12 2017. [Online]. Available: http://arxiv.org/abs/1712.09381

[101] A. Fernandes, “Indoorexplorers github repository.” [Online]. Available: https://github.com/AlexaFernandes/IndoorExplorers/tree/multi_agents

A

Code of Project

In this appendix, it is possible to find a compilation of parts of the code that are relevant to mention alongside with some definitions.

The list of requirements to install before using the IndoorExplorers environment:

Listing A.1: requirements.txt

```
colorama==0.4.4
gym==0.20.0
gym_retro==0.8.0
imageio==2.31.1
ma_gym.egg==info
matplotlib==3.3.2
numpy==1.24.3
opencv_python==4.4.0.46
opencv_python_headless==4.3.0.36
pandas==1.1.4
Pillow==10.0.0
```

```
    Pillow==10.1.0
    pygame==2.0.0
    scipy==1.5.4
    seaborn==0.12.2
    setuptools==68.2.0
    tensorflow==2.13.0
```

Example of the settings file that defines important variables, related to each agent, the creation of the map and the rendering options:

Listing A.2: settings.py

```
DEFAULT_CONFIG={
# ======== TOPOLOGY =======
#  approach (centralized or not centralized)
"approach": False, #centralized approach not implemented
#  number of agents
"n_agents": 4,
#  general configuration for the topology of operational area
"random_spawn": True, # if set to true, initial pos are ignored
# if random_spawn is set to False, then this list of pos is used to spawn each agent
    correspondently:
"initial": [ [0,0],
             [0,2],
             [3,0],
             [3,1]
            ],
"size":[16,16], #size of the map
#  configuration regarding the movements of uav
"percentage_explored": 0.9, #goal percentage of the map to be explored

# ======== ENVIROMENT =======
# configuration regarding the random map generation
# absolute number of obstacles, randomly placed in env
"obstacles":5,
# if rows/colums activated the obstacles will be placed in a semi random
# spacing
"number_rows":None,
"number_columns":None,
# noise activated only when row/columns activated
# maximum noise on each axes
"noise":[0,0],
# margins expressed in cell if rows/columns not activated
```

```
"margins":[1, 1],
# obstacle size expressed in cell if rows/columns not activated
"obstacle_size":[3,3],

# flag to activate the verification check of an agent being stuck
"check_stuck": True,
# method to check if it is stuck
    #1: count the number of steps where the agent does not discover any new cell, if it
        reaches height*width => agent is stuck
    #2: registers the last 50 positions, and if the most common one is repeated 10 times,
        then it is stuck
"stuck_method": 2,

# max number of steps for the environment
"max_steps":400,

# ======== REWARDS ===========
"movementCost": 10, #this is discounted for every time step/every movement made (don't put
    the minus sign!!)
"new_cell_disc_reward": 10, #reward value for each new cell discovered
"bonus_reward": 1000, #reward for exploring "percentage_explored"% of the map
"stuck_reward": -1000, #penalty for getting stuck between positions
"collision_reward":-1000, #penalty for colliding with walls
"out_of_bounds_reward":-1000, #penalty for going out of the bounds of the map


# ======== SENSORS | LIDAR =======
"lidar_range":3, #defines the LiDAR range
"lidar_channels":32,

# ======== COMMUNICATION =======
"comm_range": 3.0, #defines the communication range of each agent

# ======== VIEWER =========
"viewer":{"width":21*30,
          "height":21*30,
          "title":"Indoor-Explorers-V01",
          "drone_img":'/home/thedarkcurls/IndoorExplorers/img/drone.png',
          "obstacle_img":'/home/thedarkcurls/IndoorExplorers/img/stone_black2.png',
          "background_img":'/home/thedarkcurls/IndoorExplorers/img/wood_floor.jpg',
          "light_mask":"/home/thedarkcurls/IndoorExplorers/img/light_350_hard.png",
          "night_color":(20, 20, 20),
          "draw_lidar":True, #not used
          "draw_grid":True, #not used
          "draw_traceline":False, #not used
```

```
            "print_map": True, #enables the visualization of the map of each agent
                individualy, when rendering is active
            "print_prompts": False #enables the visualization of the map of each agent
                individualy, when rendering is active
          }
}
```

Definition of the agent class:

Listing A.3: Agent class

```
class Agent(object):
    def __init__(self):
        super(Agent, self).__init__()
        self.name = ''
        self.id = None
        self.pos = None #position
        self.reward = 0
        self.done = False
        #each agent has its own explored map
        self.exploredMap = []
        self.pastExploredMap = []
        #flag to signal stuck status
        self.stuck = 0
        #flag for collisions
        self.collision = False
        #flag for out of bounds status
        self.out_of_bounds = False
        #communication range
        self.c_range = 3.0
        # communication noise amount
        self.c_noise = None #not implemented
        # color
        self.color = None
        # state
        self.state = None
        # action
        self.action = None

    #it's considered in range, inside a square with distance of c_range squares around the
        agent
    def in_range(self, agent2):
        delta_pos = abs(np.subtract(np.array(self.pos), np.array(agent2.pos)) )
        if delta_pos[0] > self.c_range*2 or delta_pos[1] > self.c_range*2 :
```

```
            return False
        else:
            return True

    #checks if the agent has not collided or out of bounds
    def is_alive(self):
        return (not self.collision) and (not self.out_of_bounds)
```

A basic example of an OpenAI Gym code:

Listing A.4: Example of a basic OpenAI gym code

```
if __name__ == "__main__":
    #creation of the environment
    env = gym.make('indoor_explorers:exploConf-v01')

    #reset environment
    observation = env.reset()

    #for 100 episodes:
    for _ in range(100):
        env.render() #render environment

        #choose action
        action = env.action_space.sample() #your agent here (in this case takes random
            actions -> the goal is to use DDDQN)

        #apply action inside the step function
        observation, reward, done, info = env.step(action, True)

        #while the agent is not done the episode does not end
        if done:
            observation = env.reset() #once it is done, it resets
        time.sleep(0.3) #add a little waiting time for visualization

    env.close()
```

B

Larger figures

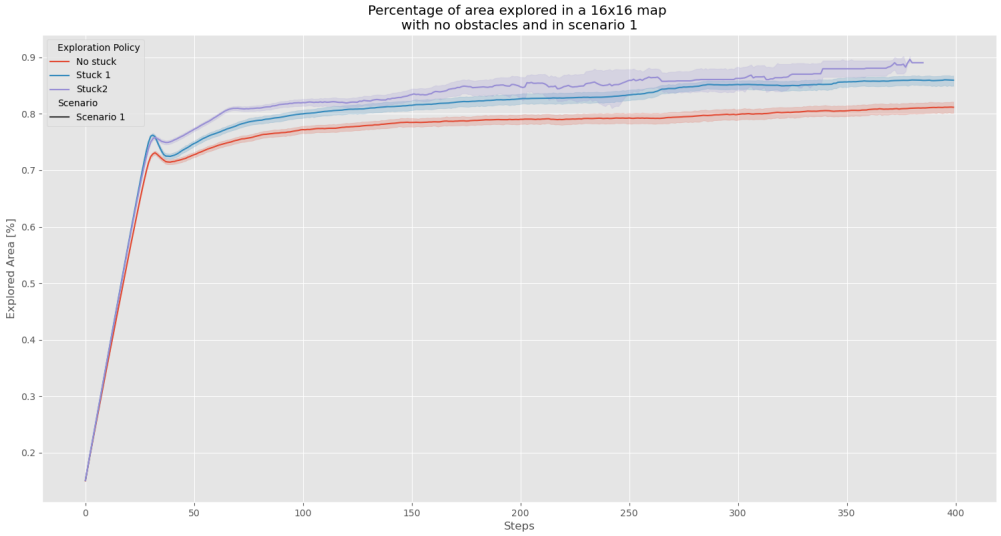


Figure B.1: [Fig.4.2 zoomed out] Percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles in scenario 1

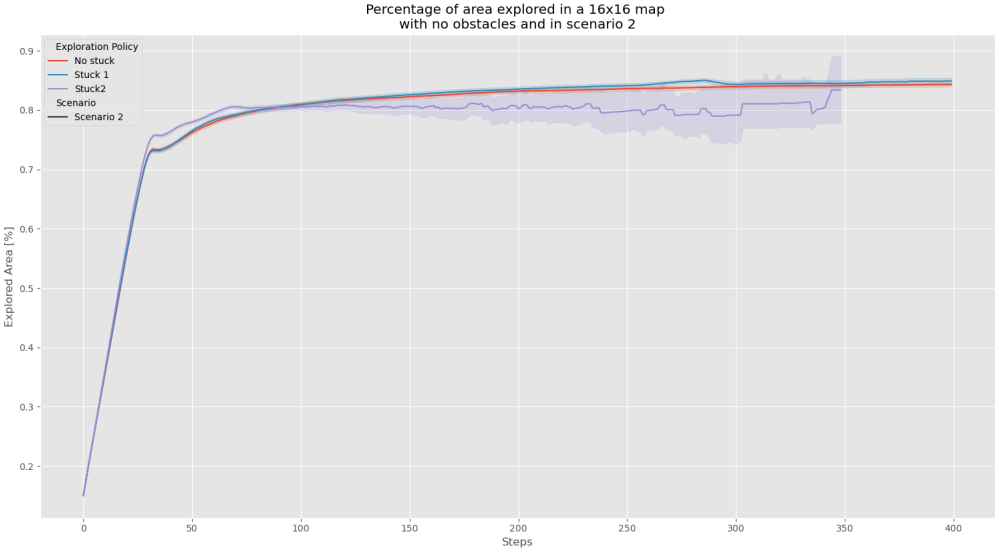


Figure B.2: [Fig.4.5 zoomed out] Percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles in scenario 2

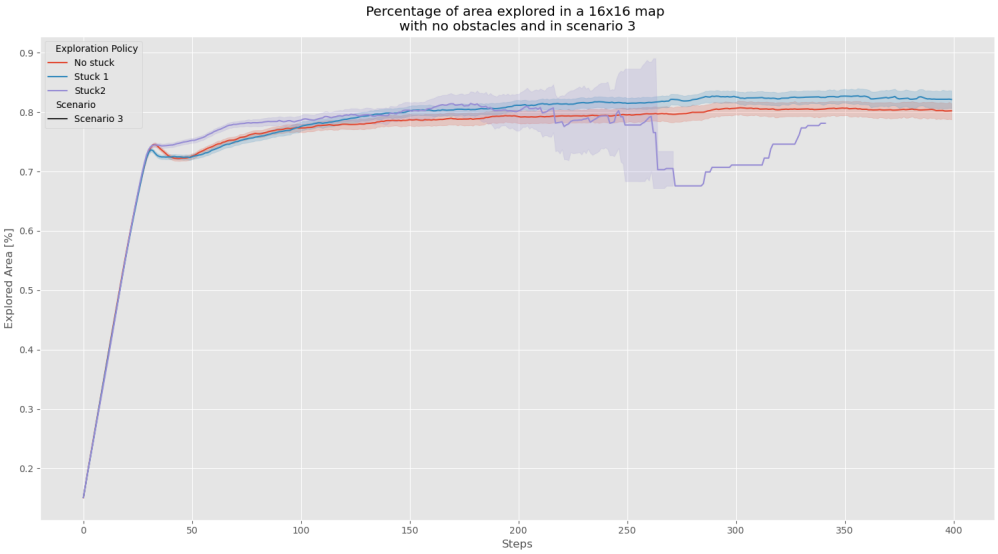


Figure B.3: [Fig.4.7 zoomed out] Percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles in scenario 3

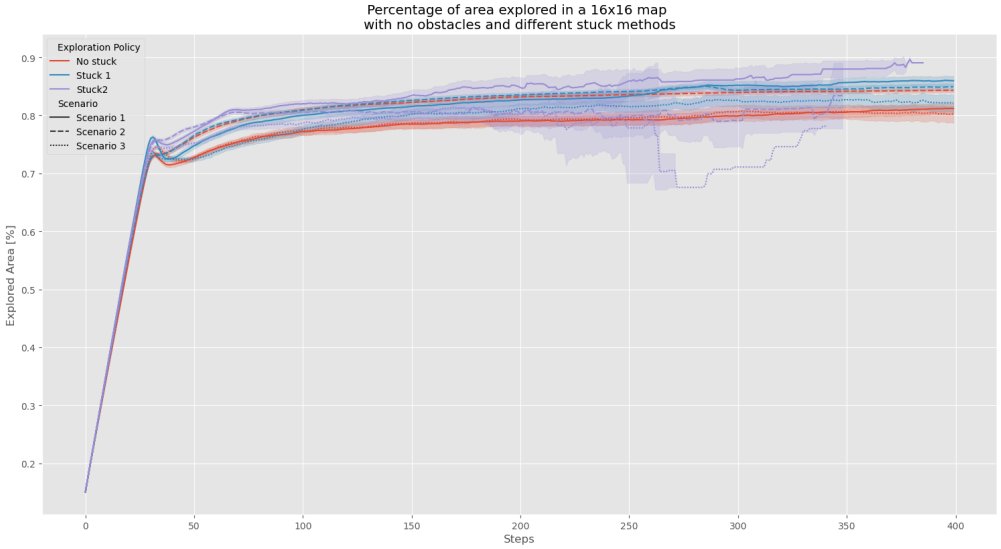


Figure B.4: [Fig.4.8 zoomed out] Overview of percentage of the area explored with respect to the number of steps, in a 16x16 area with no obstacles

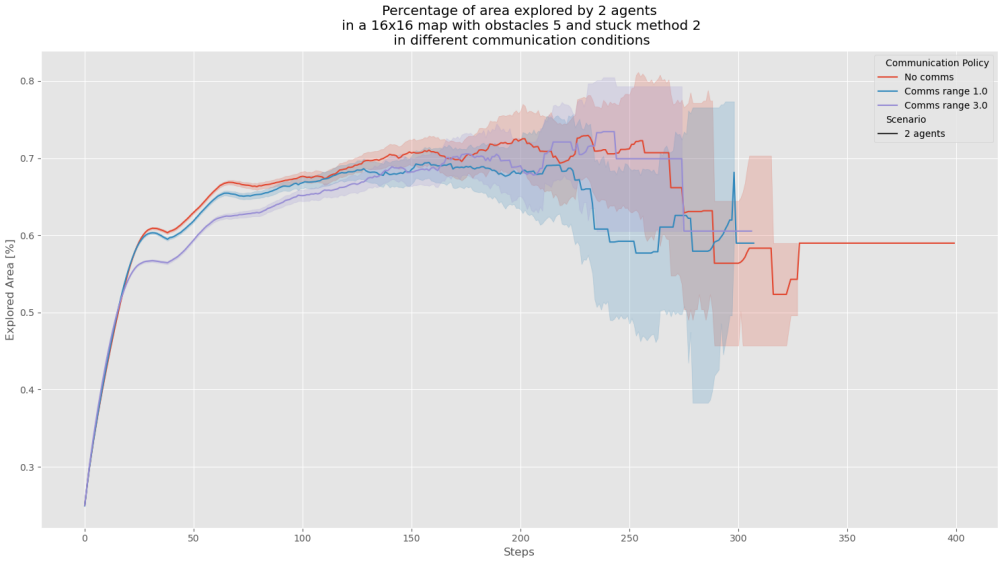


Figure B.5: [Fig.4.16 zoomed out] Percentage of the area explored with respect to the number of steps, by 2 agents various communication ranges in a 16x16 area with 5 obstacles

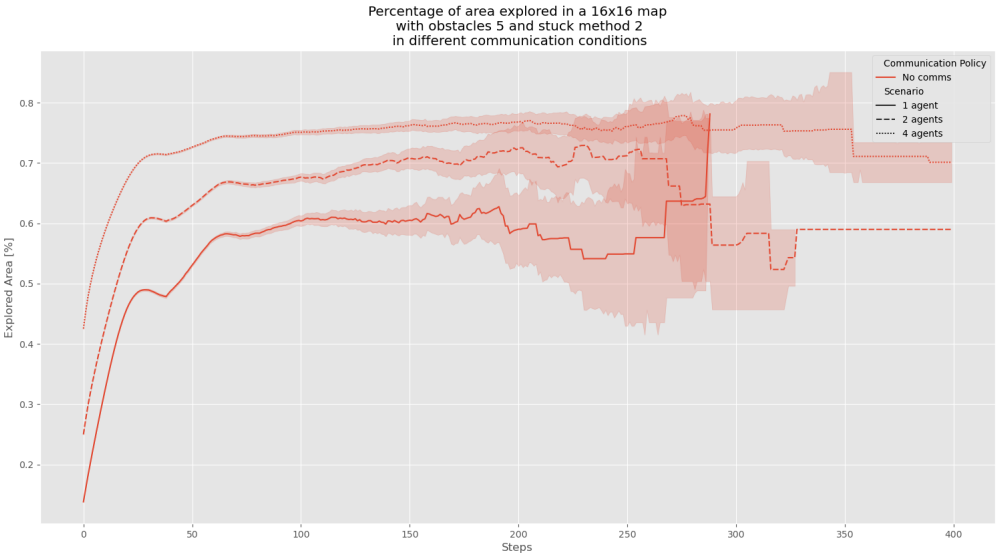


Figure B.6: [Fig. 4.19 zoomed out] Percentage of the area explored with respect to the number of steps, by various number agents and no communication ranges in a 16x16 area with 5 obstacles

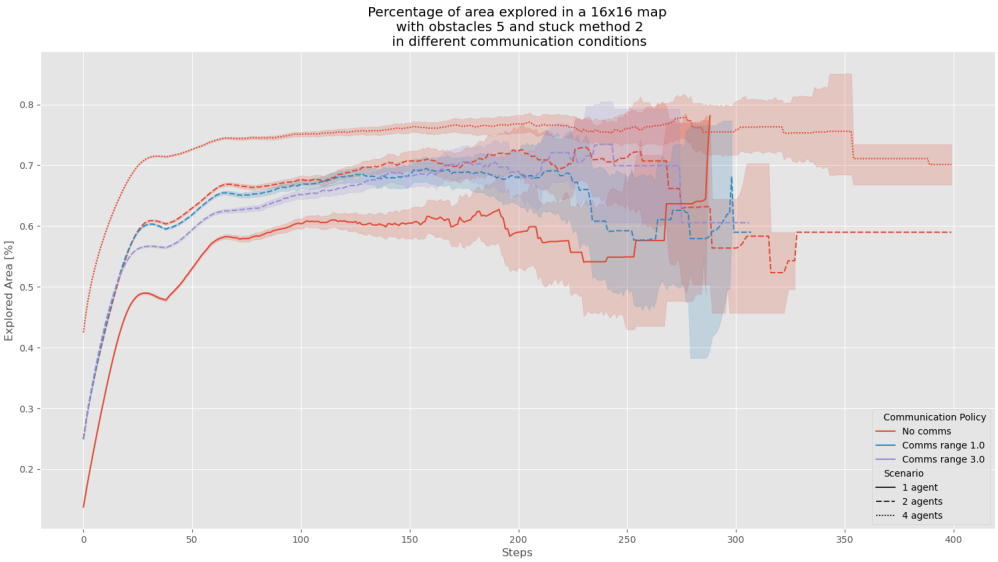


Figure B.7: [Fig.4.20 zoomed out]Overview of the percentage of the area explored with respect to the number of steps, in a 16x16 area with 5 obstacles