Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2019

Diogo Vidal e Silva

Navegação de um Robô Móvel Usando Aprendizagem por Reforço

Mobile Robot Navigation Using Reinforcement Learning

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2019

Diogo Vidal e Silva

Navegação de um Robô Móvel Usando Aprendizagem por Reforço

Mobile Robot Navigation Using Reinforcement Learning

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Dr. Filipe Miguel Teixeira Pereira da Silva, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Dr. Vítor Manuel Ferreira dos Santos, Professor Associado c/ Agregação do Departamento de Engenharia Mecânica da Universidade de Aveiro

agradecimentos / acknowledgements

Gostaria de agradecer todo o apoio, ajuda e tempo dispendido durante a realização desta dissertação ao Professor Filipe Silva e ao Professor Vítor Santos e por me terem dado o privilégio de conhecer de perto a área do Machine Learning, em especial o Reinforcement Learning.

Tenho também que agradecer a toda a minha famlia, em especial aos meus pais pelo constante apoio nas fases boas e más que passei ao longo destes cinco anos de Universidade e por me darem a possibilidade de tirar o curso com que sempre sonhei formar-me.

Por último mas não menos importante quero agradecer a todos os meus colegas, em especial a minha namorada, que me acompanharam desde o primeiro dia até ao último desta caminhada, mostrando sempre uma forte cumplicidade e entreajuda de forma a superar todos os desafios que nos foram propostos.

Palavras-Chave

Navegação Robótica Móvel; Aprendizagem por Reforço; Estrutura Hierárquica; Q-Learning; Ambiente Tipo Labirinto

Resumo

Há um interesse crescente no desenvolvimento de tecnologias de serviços e robôs de assistência para aplicação em ambientes domésticos e urbanos. Entre as habilidades necessárias estão navegação autónoma e manutenção de segurança. O Machine Learning fornece um conjunto de ferramentas computacionais que se mostraram úteis para a navegação de robots, como redes neuronais, Reinforcement Learning e, mais recentemente, Deep Learning. Esta dissertação tem como objetivo investigar o problema da navegação de um robot móvel num labirinto utilizando Reinforcement Learning. Em particular, o trabalho concentra-se em dimensionar o Reinforcement Learning, e o Q-learning em particular, para um problema do mundo real usando um robot físico. Primeiro, para evitar grandes espaços de estado-ação, o sistema robótico é treinado usando uma abordagem hierárquica na qual componentes de baixo nível (sub-tarefas) são sequenciados num nível superior. Em segundo lugar, uma função de reward consistente é projetada para a navegação do robô num corredor e num canto, fornecendo ao robot mais informações (conhecimento prévio) após cada ação. As experiências conduzidas, utilizando um robot simulado e real, mostram a viabilidade da abordagem hierárquica reduzindo a complexidade da tarefa de aprendizagem e o papel da função de recompensa na especificação de um objetivo. Finalmente, o estudo providencia uma avaliação detalhada sobre a experiência transferida de simulação para o robô físico.

Keywords Mobile Robot Navigation; Reinforcement Learning; Hierarchical Structure; Maze; Q-Learning; Maze-like Environment

Abstract

There is a growing interest in the development of service and assistive robot technologies for application in domestic and urban environments. Among the required abilities are autonomous navigation and safety maintenance. Machine Learning provides a set of computational tools that have proved useful for robot navigation, such as neural networks, reinforcement learning and, more recently, end-to-end deep learning. This dissertation aims to investigate the problem of mobile robot navigation in a maze-like environment using a reinforcement learning framework. In particular, the work focuses on how to scale reinforcement learning, and Q-learning in particular, to a real-world problem using a physical robot. First, in order to avoid large state-action spaces and long horizons, the robot system is trained using a hierarchical approach in which low-level components (sub-tasks) are sequenced at a higher-level. Second, a dense reward function is designed for robot navigation in a corridor and moving around a corner, providing the robot with more information (prior knowledge) after each action. The experiments conducted, using a simulated and a real robot, show the feasibility of the hierarchical approach in reducing the complexity of the learning task and the role of the reward function in goal specification. Finally, the study provides detailed evaluation about transferring experience in simulation to the physical robot.

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
EPFL	Ecole Polytechnique Fédérale de Lausanne
GPIO	General Purpose Input/Output
HAM	Hierarchies of Abstract Machines
HRL	Hierarchical Reinforcement Learning
HSMQ	Hierarchical Semi-Markov Q-Learning
IRL	Inverse Reinforcement Learning
IR-LED	Infrared Light Emitting Diode
MDP	Markov Decision Processes
ML	Machine Learning
PCB	Printed Circuit Board
PSD	Position Sensitive Detector
RL	Reinforcement Learning
ROS	Robot Operating System
SMD	Surface-mount Technology
SMDP	Semi-Markov Decision Processes
SOC	System On Chip
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
THT	Through-Hole Technology
UDP	User Datagram Protocol

Chapter 1

Introduction

1.1 Background and Context

Technology is in constant evolution, being that in the last decades its growth is largely exponential. Everything around us has become simpler thanks to advances with technology in industry, agriculture, health and even in our lifestyle. Our grandparents and great-grandparents, at their youngest age, did not have the privilege of having a television at home, the internet or even a personal cell phone. This temporal difference between then and now serves as a reminder of how prevalent technology is in our lives today, more often than before, the words Artificial Intelligence (AI) and Machine Learning (ML) are heard as they contaminated the media. There are self-driving cars, robots that perform tasks without any supervision that were previously performed by humans and even mobile robots that can navigate in a real environment without colliding with the environment.

A human being has an extreme need to know how he/she thinks and how he/she acts in a thoughtful way. AI tries to understand and build intelligent entities [1]. Human intelligence is mimicked by a machine that, through learning processes and taking into account the information acquired, is able to approximate the behavior of its counterpart. The tasks most studied by AI include voice recognition, autonomous driving, medicine, space exploration and finance. In the 1950s a generation of mathematicians and scientists brought AI to life. One of them, the famous British mathematician Alan Turing proposed that just as the human being was able to make decisions and solve problems, maybe the machines could do it too [2]. After 70 years, AI is present in almost everything around us and only the future will know what is more to come.

ML is a branch of Artificial Intelligence that uses algorithms which allows one to create mathematical models through acquired information to improve predictive and decision-making power without the computer being explicitly programmed to accomplish this task. As defined by T. Mitchell, "The computer program is said to learn from experience and with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." [3]. Machine Learning is often divided into three categories: Supervised and Unsupervised Learning and Reinforcement Learning.

In Supervised Learning, it is given a training set with labeled data that divides the totality of data into different categories. This is used for classification or regression problems, with each entry having n features and the corresponding label. A classification problem is intended to identify an input value as belonging to one of the output classes. Suppose for example that the dataset is composed of correctly labeled images of dogs, cats and rats. After training and receiving new input, the algorithm should be able to know what animal each image is. A regression problem outputs a real value, and not a class, aiming to get a forecast given a certain input value. This is widely used, for example, to predict real estate prices, weather forecast, et cetera. Unsupervised Learning intends to create a model capable of predicting the category to which an example corresponds, receiving a training set without labels. More specifically, the information is not previously divided and the algorithm, taking into account the features of each example, groups them. This branch of Machine Learning is widely used in market/consumer segmentation for example.

Reinforcement Learning (RL) can be considered another branch of ML that does not require previous data, instead using its experience to explore the environment to produce the best end results. Consider, for example, attempting to train a mobile robot to reach a goal when navigating a maze-like environment. The process ends up being simple: an agent intends to reach a goal where the path to it is totally unknown using actions previously carried out. Upon performing the actions for the first time, the agent creates many paths to the goal which form a map of the environment with all the states and the best actions to follow in each of them. The agent is then able to use those previously completed actions and discern which path will result in the entire task being completed with the maximum possible reward. In other words, a RL framework requires that the programmer specifies **WHAT** to do and the robot learns **HOW** to perform the task by trial-and-error interaction with the environment.

This dissertation focuses mainly on the latter example: a hierarchical reinforcement learning approach will be used to allow a mobile robot to navigate autonomously in a maze in order to reach its end without colliding with the environment and choosing the optimal path to the goal.

1.2 Motivation and Objectives

This work deals with the problem of mobile robot navigation in a real-world maze-like environment using a Reinforcement Learning approach. The particular focus of the work is to analyze a simple case in some detail to demonstrate how Reinforcement Learning can be applied to solve a real-world problem. Reinforcement Learning is a computational approach that enables a robot agent to find an optimal behavior through trial-and-error interaction with the environment. However, applying RL to robotic problems is, generally, a difficult problem with many challenges [4].

The main challenges in robot RL include coping with large state-action spaces and long horizons, sample inefficiency given that learning a policy may need an impractical number of interactions (reducing often its application to simulation), the difficulty to specify a reward function that exactly capture the problem at hands, the occurrence of local optima and the difficulty to escape from them, the need for parameterized function approximators (e.g., neural networks) in order to generalize between similar situations and actions, mainly in case of large, complex, high-dimensional environments.

A promising approach to reduce the complexity of the learning task is to hierarchically decompose the problem into basic sub-tasks which are coordinated by a higher level. In this context, Hierarchical Reinforcement Learning (HRL) seems to be particularly well-suited for navigation in a maze in order to reduce the search space and speed up the learning process. HRL switches some complexity of the navigation task to a lower-level of functionality, where local policies are required to learn simple sub-tasks such as moving in a corridor, getting around a corner or crossing-a-door. These locally learned policies will then be selected by a higher-level policy in order to achieve the specific goal. Hopefully, they can also be reused to more complex tasks and/or adapted to new situations. Although a faster learning can be expected, a robotic simulator will assume a key role at the programming environment for developing the robot behavior, such that the algorithms is tested under numerous reproducible conditions.

Having this in mind, the main objectives of this dissertation can be described as follows:

- To develop the hardware and software infrastructures required to operate the mobile robot in a real maze environment.
- To investigate how the introduction of hierarchy into the problem can contribute to make the navigation task tractable.
- To evaluate the possibility of transfer the experience gained in simulation to a real robot navigating in a similar maze environment.

1.3 Dissertation's Structure

This master dissertation is composed of six chapters, including the introduction:

- Chapter 2 describes the related work previously done using Reinforcement Learning, focusing on the hierarchical approach in mobile robot navigation as well as a theoretical background of the Reinforcement Learning.
- Chapter 3 describes the experimental setup of both the Hardware (robot, sensors and communication module) and Software (IDEs, ROS, Simulator) used during this master thesis.
- Chapter 4 presents the methodology proposed to solve the problem at hand is depicted. It is described the division of the maze in sub-tasks, the algorithm used and the hierarchical approach using a topological map.
- Chapter 5 shows the experiments made and the results obtained in each one of the sub-tasks and how the whole maze is used in both simulation and the real environment.
- Chapter 6 presents the conclusions of this thesis and the future work related to the scope of this work.

Chapter 2

Theoretical Framework

Over the years, robotics has become a very challenging case study and previous thought was that it would be possible to teach a robot to do activities a human being can; mainly repetitive day-to-day activities where a robot could replace the human being. Instead, the idea of a completely autonomous robot is not something that is possible in the present, nor do we know if it is would be a beneficial vision for human beings. Machine Learning, as a field of Artificial Intelligence, is one of the main techniques in the learning process within robots to increase their autonomy. It is already possible to see car companies with cars able to function using auto-pilot thanks to Machine Learning and Computer Vision [5]. It is used in many fields, especially robot vision [6, 7], mobile robot navigation [8, 9] and medical and surgery application [10]. A robot can learn from the information given by a dataset, known as Supervised and Unsupervised Learning, or can learn through the experience obtained with the interaction with the environment, Reinforcement Learning.

This chapter summarizes the research conducted at the beginning of this dissertation with the objective of gaining insight into Reinforcement Learning, mobile robotic navigation and the state of the art of model free and model based methods as well as the hierarchical approach to reinforcement leaning.

2.1 Mobile Robot Navigation

A robot is an autonomous system that always has a main objective and to act requires sensory information that allows it to correctly fulfill these objectives. Navigation is the ability to understand the current position and to be able to plan a path towards some goal location. It is indispensable for the mobile robot to always know where it is in the environment to know what action to take. Navigation can be defined by a set of blocks (Fig.2.1).

Leonard and Durrant-Whyte divided the problem of mobile robot navigation into three central points [11]:

- Where am I? - addresses the problem of robot location. Through sensory information it is possible to infer the location, or a restricted set of locations, where the robot can be found.

- Where I am going? - addresses the problem of mapping. When the robot navigates through unknown zones, it can create a map of the environment using the information that the sensors offer.

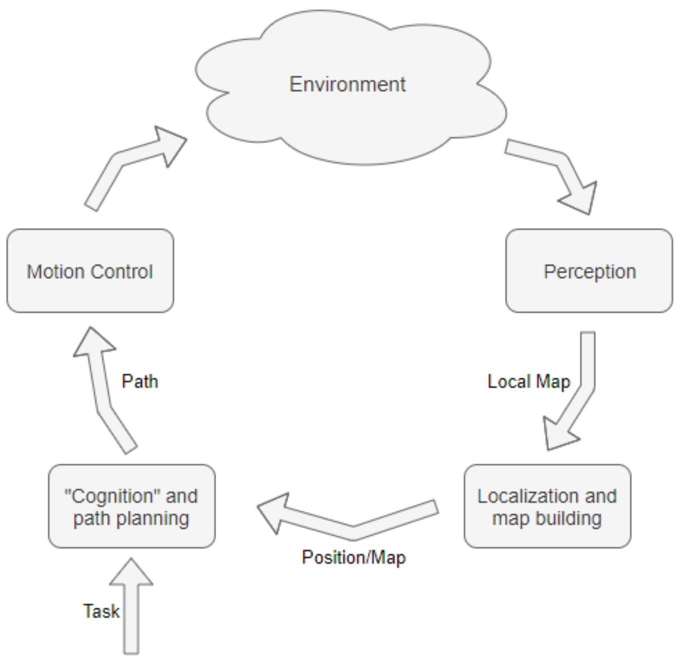


Figure 2.1: Building blocks of mobile robot navigation.

- How do I get there? - addresses the problem of path planning. The robot aims to move from a position of the environment to its destination and for this you need to plan your route in the best possible way.

In order for a robot to navigate, it needs a set of elements:

- A mechanism that allows it to move around the environment.
- Sensors for information about its surroundings.
- A system where the information received by the sensors is processed and where the action to be taken is chosen.
- A system that allows controlling the wheels in order to perform the actions.

2.1.1 Mobile Robots

Mobile robots have the peculiarity of being able to move around unknown environments using a locomotion mechanism such as a pair of wheels, a set of legs or propellers. It is a system with mobility, a certain level of autonomy and perception of the environment. Within mobile robots there are three main groups:

- UGV - Unmanned Ground Vehicles that consist of wheels, tracks or legs - These vehicles are widely used for the transportation of material in agriculture, making it possible to harvest and regulate concentrations of chemicals such as sulfates and herbicides in plantations. They are also commonly used by military services in rescue missions, fires, mine detection and detonation, and also in war scenarios to reduce military contact with the enemy reducing the number of casualties. Another application of these vehicles is the space exploration where companies like NASA built autonomous vehicles to be sent in the expeditions in order to know more about the properties of the other planets and moon [12, 13].

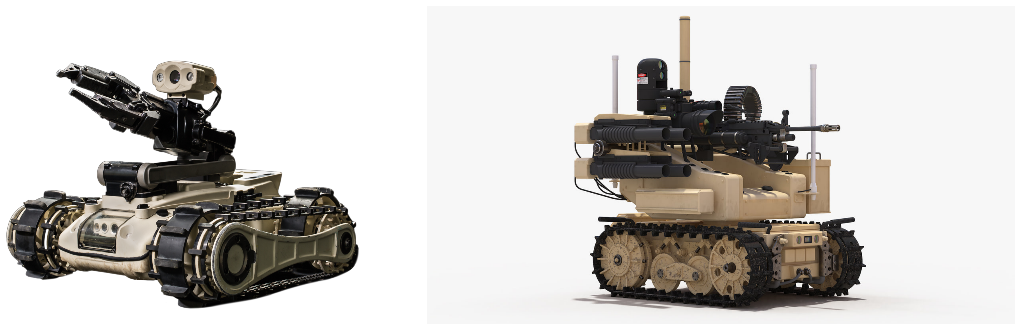


Figure 2.2: MTGR (left) and Talon Swords (right) - Lightweight Combat Proven Tactical Robot.

- UAV - Unmanned Aerial Vehicles - These vehicles are widely used by the military in reconnaissance and surveillance missions and anti-terrorism campaigns, as well as attacks on the enemy. They are also used in the distribution of goods by large companies, in the filming of movies and journalistic reports, in agriculture and in the resolution of disasters such as fires and earthquakes [14, 15].

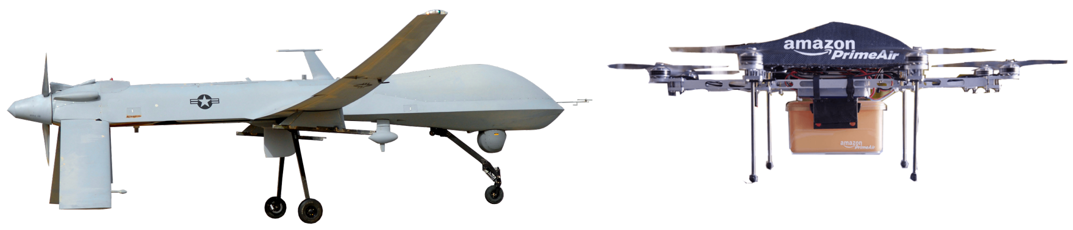


Figure 2.3: MQ-1 Predator - Military service (left) and Amazon Prime Air - Delivery System (right).

- AUV - Autonomous Underwater Vehicles - These vehicles are widely used to make deep sea expeditions to find new marine species or debris from ships or aircrafts that may have sunk/crashed and to obtain sensory information such as pH and the presence or absence of light, for example. They are also commonly used by companies in the oil industry to create maps of the oceanic surface before the construction of the infrastructure and by the military in missions of surveillance and reconnaissance and mining countermeasures for example [16, 17].

Figure 2.4: Pluto Plus - countermining (left) and WHOI SeaBED - collect sonar and optical images of the seafloor (right).

2.2 Reinforcement Learning

RL is a branch of Machine Learning with a somewhat different concept from the other approaches. The agent of the problem is constantly moving and producing actions. This causes it to create interactions with the environment by trial and error, receiving feedback as a reward from these actions. The main objective of the agent is to reach the goal using the minimum number of possible actions while gaining the maximum reward possible. This reward may or may not be immediate, and the possibility of obtaining a delayed reward in conjunction with the trial and error approach is what makes Reinforcement Learning so special.

It is expected that during the learning of a task, the agent will often fail at the beginning stages. As the interaction with the environment increases, the behavior improves. This provides the opportunity at the end of the learning process to perform the task without errors following the policy that generates the correct action for each moment.

Reinforcement Learning can be seen as an analogy to our lives as humans. At birth, we basically have no knowledge and do not know how to perform all tasks that life puts before us. However, with the passage of time and experience, we are able to act according to the problem that we have to overcome. Similar to a human being, a robot can also learn with the help of Reinforcement Learning, requiring only a mean that makes it possible to collect information from the environment. The theoretical framework for reinforcement learning is based on the book by Richard Sutton and Andrew Barto [18].

2.2.1 Elements of Reinforcement Learning

The main elements of reinforcement learning are the agent and the environment but these alone do not give results for the task to be performed, so there are four sub-elements that must be present: a policy, a reward signal, a value function and a model of the environment.

- The policy shows the action the agent must perform, given the state the agent is in within the environment, producing a map of the environment with all the existing states and the corresponding action that produces the best result. It is, ordinarily, a stochastic process.

- A reward signal is what defines the purpose of a reinforcement learning problem. At each new iteration, the agent receives a reward from the environment after the action, and the purpose of the agent is to maximize the sum of rewards for each episode. This reward shows the agent whether the action taken is good or bad in relation to the ultimate goal. The analysis of these rewards is what builds the policy, which throughout the episodes is changed so that in the end it corresponds to the optimal policy.

- A value function of a state is the expected sum of rewards that the agent can expect until the goal starting in that state.
- The last sub-element is the environment model, which allows replication of the environment and the extraction of important information about behavior. A template is used in particular in the planning phase, which assists in the decision making of a next action or reward. Within the models are two distinct types: model-free which are basically trial and error and model based methods that use models and planning.

2.2.2 Markov Decision Processes

Markov Decision Processes (MDP)s are a bridge to solving a reinforcement learning problem that are defined by:

- An environmental state, $S_t \in \mathcal{S}$, where $\mathcal{S}$ designates the state space.
- The action, $A_t \in \mathcal{A}$, where $\mathcal{A}$ designates the action space.
- A numerical reward, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$.
- The state transition probabilities, that show what is the probability being in State S_t and applying action A_t to go to new state s'.

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

- The reward function for state-action-next state triples as a three-argument function $r : \mathcal{S} * \mathcal{A} * \mathcal{S} \to \mathbb{R}$.

$$r(s, a, s') \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r\in\mathcal{R}} r\frac{p(s', r|s, a)}{p(s'|s, a)} \tag{2.2}$$

The MDP framework then proposes that a problem with a given objective be reduced to three signals between the agent and the environment: actions taken by the agent, possible states and rewards (Fig. 2.5).

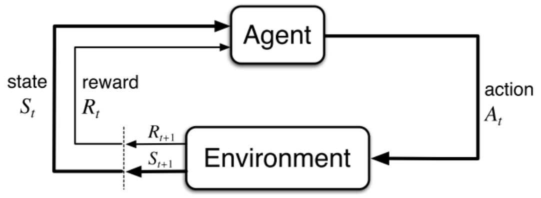


Figure 2.5: The agent-environment interaction.

2.2.3 Policy and Value Function

A policy gives the agent the probabilities, in each state, of selecting the possible actions. This policy is constantly changing during the tests. The value function of a state under a policy π, designated *v_π(s)* is the expected return when starting in state s and using π.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\Big[G_t|S_t = s\Big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|S_t = s\Big] \\ &= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)\Big[r + \gamma\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big] \end{aligned} \tag{2.3}$$

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s'), \forall s \in \mathcal{S}] \tag{2.4}$$

The equation 2.4 is the Bellman equation for the policy value function. There is also an action value function under a policy π, designated *q_π(s,a)* that shows the expected sum of rewards while taking action a in state s.

$$q_\pi(s,a) = \mathbb{E}_\pi\Big[G_t|S_t = s, A_t = a\Big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|S_t = s, A_t = a\Big] \tag{2.5}$$

Where G$_t$ is the return, that is, the sum of the rewards.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \tag{2.6}$$

As MDP is normally used for finite cases, each interaction between agent and environment has a final state, which shows the moment when the agent reaches the goal. Each time the agent reaches the target, a new episode begins, resetting the position by sending the agent to an initial state. γ is the discount rate, value between 0 and 1, which quantifies the difference in importance between immediate and future rewards. If γ is a value near 1, the future rewards are taking more seriously. If $\gamma = 0$, the agent takes into account only the immediate reward.

2.2.4 Optimal Policy and Optimal Value Function

When the return of a policy π is greater than or equal to that of a policy π' in all states this policy is called an optimal policy. There is the possibility of a set of optimal policies, π_*, however they all have the same state-value function, defined as v_*, and the same optimal action-value function, defined as q_*.

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in S. \tag{2.7}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in S \ \Lambda \ a \in A. \tag{2.8}$$

The value function, v_*, needs to satisfy the conditions of the Bellman equation, 2.4. Being the optimal value function does not need to specify the policy that concerns, giving itself the name of *Bellman optimality equation.* This equation, 2.10, shows that the value of a state must be equal to the expected return for the best action in that state for the optimal policy case:

$$\begin{aligned}
v_*(s) &= \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}\big[G_t|S_t = s, A_t = a\big] \\
&= \max_a \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a\big] \\
&= \max_a \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a\big]
\end{aligned} \tag{2.9}$$

$$= \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big] \tag{2.10}$$

It is also possible to demonstrate the Bellman optimality equation for the optimal action-value function:

$$\begin{aligned}
q_*(s,a) &= \mathbb{E}\big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1},a')|S_t = s, A_t = a\big] \\
&= \sum_{s',r} p(s',r|s,a)\big[r + \gamma \max_{a'} q_*(s',a')\big]
\end{aligned} \tag{2.11}$$

The Bellman optimality equation in finite MDPs has only one solution that is policy independent. Having n states, there are in fact n equations with n unknowns. If the state transition probabilities, p, are known it is possible to solve the system of equations in the case of v_* and consequently to q_*.

It is simple to find the optimal policy from v_*, since it follows a greedy approach to the optimal value function v_*. The optimal policy chooses the action to be taken, taking into account the consequences in the near future. However, as the optimal value function takes into account the whole process, the optimal policy works in the long term as well.

It is also possible to choose the optimal actions through the optimal action-value function, q_*, being an even simpler process because it is not necessary to do a one-step-ahead search for a state. It is only necessary to know which action maximizes $q_*(s,a)$. Thus, the optimal action-value function provides the ability to find the optimal action without knowing anything about the previous states, that is, about the dynamics of the environment.

2.3 Solution Methods

Model-based methods acquire information on the probability transitions to generate a model, which is used to plan the future iterations. Unlike the model-free methods, the duration of interaction with the real environment during the learning phase is very small since after the creation of a model the whole learning process can be done in simulation. Although accelerating the learning process, the use of this method may not be convenient since it can provide inaccurate results if the model is not well constructed and it is also a method with a very high computational level.

Model-free methods do not require an environment model to perform their function, learning the optimal policy solely by interacting with the environment. The agent performs an action and receives a reward from the environment that can be positive or negative, and the goal is to select the actions that give the the highest cumulative sum of reward. Over time, the agent will explore the environment and learn the best action to take in view of the state in which it is. The best-known model-free methods are divided into two strands: Monte Carlo Methods and Temporal Difference Learning which will be explained below.

2.3.1 Dynamic Programming

In order to calculate a value function using *Dynamic Programming* it is necessary to have a model of transition probabilities T(s',a,s) and the reward function R(s,a). This information makes it possible to calculate the value of a state using the expectation of the next reward and the next value. In simple problems, *Dynamic Programming* can provide the exact solution of the value function using an iterative approach, however due to its extremely high computational cost it is rarely used because it requires a perfect environment model. In most cases it only gives exact solutions to problems with discrete state space and action spaces. The most well-known algorithms within the *Dynamic programming* branch are *policy iteration* and *value iteration.*

In 1957, Bellman showed that these methods could be applied to a lot of problems coined it *Dynamic Programming* [19].

2.3.2 Monte Carlo Methods

Monte Carlo methods only need experience, interacting with the environment throughout the learning process. This learning is divided into episodes, each episode ending in the arrival of the agent to the goal and only then is the policy and the function of value is estimated.

These methods follow the GPI (generalized policy iteration) scheme, allowing evaluation and improvement of policy. Instead of using a template to calculate the value of each state, it does the average of returns that started in that state. In order to obtain a reliable optimal policy in a simulated environment it is necessary for each episode to begin learning at a random position so as to have sufficient information for all state-action pairs. If it is in a real environment and this random start is not possible there are two possibilities, On-Policy Monte Carlo Control and Off-Policy Monte Carlo Control. The difference between the two is that, in the case of the On-Policy method, the agent is constantly exploring the best possible policy, while in the Off-Policy method the agent explores and at the same time learns an optimal policy different from the one use.

2.3.3 Temporal Difference Learning

Temporal-Difference Learning is a combination of the *Dynamic Programming* and *Monte Carlo* methods, since it does not know the dynamics of the environment by learning only through the experience gained by interacting with it as well as in the *Monte Carlo* method and updating estimates of the value function based on other estimates as in the case of *Dynamic Programming.* Basically the agent chooses an action according to a policy interacting with the environment. When looking at the new state updates the value of the current state taking into account the learning rate factor, α, as shown in Equation 2.12.

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma * V(s_{t+1}) - V(s)] \tag{2.12}$$

TD-Learning has certain advantages over Dynamic Programming and Monte Carlo methods. Compared to Dynamic Programming, it does not require an environment model gaining only experience through visits to the possible states. It is also more advantageous than the Monte Carlo Methods because learning is done online and does not require the end of an episode to estimate the value function.

The most well-known algorithms in this field of Reinforcement Learning are Q-Learning and SARSA and as this approach will be used throughout this dissertation both will be explained in more detail.

Q-Learning

Q-Learning is one of the best-known and discussed methods of Reinforcement Learning being a model-free off-policy temporal difference method. It was idealized and introduced by Watkins in 1989 [20] and in 1992 it was rigorously proven by Watkins himself and Dayan [21]. This method allows the agent to learn to act optimally by experiencing the consequences of actions without having to create a map or model of the environment and is defined by the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.13}$$

The action-value function, Q, is a direct approximation of the optimal action-value function, q_*, being totally independent of the policy followed allowing a very fast convergence through a simple algorithm. In Q-Learning the policy is used to determine the state-action pairs that are visited and updated. This update allows for correct convergence.

Algorithm 1 Q-Learning Algorithm.

```
 1: Initialize Q(s,a) arbitrarily
 2: repeat {for each episode}
 3:   Initialize s
 4:   repeat {for each step of episode}
 5:     Chose an action, a, from the state, s, using policy derived from Q (e.g. e-greedy);
 6:     Take action, a, and observe reward, r, and state, s'
 7:     $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * \underset{a'}{max} Q(s',a') - Q(s,a)]$
 8:     $s \leftarrow s'$
 9:   until s is terminal
10: until maximum episode
```

SARSA (State-Action-Reward-State-Action)

SARSA is a model-free on-policy temporal difference control method. It was first introduced by Rummery and Niranjan in 1994 under the name of *modified Q-Learning* [22] and later Sutton gave it the name SARSA [23]. It is a method very similar to the method of Q-Learning, but the difference is that the maximum reward for the next step is not necessarily used to update the value of Q and the new action and reward is selected again through the policy initially used (e.g. e-greedy). The action-value function in this method is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \Big] \tag{2.14}$$

The name SARSA comes from this quintuple *Q(s,a,r,s',a')* that allows the agent to perform the update, where s and a are the original state and action, r is the reward observed, *s'* and *a'* is the new state pair-action.

In an on-policy method, the action-value function Q^π is continuously estimated for behavior policy π, changing this policy at the same time in order to choose the best action. The convergence of this method depends on the nature of the policy's dependence on Q, being e-greedy or e-soft for example. As state-action pairs are visited endlessly and policy converges towards a greedy policy, SARSA converges for optimal policy and action-value function.

Algorithm 2 SARSA Algorithm.

```
1:  Initialize Q(s,a) arbitrarily
2:  repeat {for each episode}
3:    Initialize s
4:    repeat {for each step of episode}
5:      Chose an action, a, from the state, s, using policy derived from Q (e.g. e-greedy);
6:      Take action, a, and observe reward, r, and state, s'
7:      Chose an action, a', from the state, s', using policy derived from Q (e.g. e-greedy);
8:      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * Q(s',a') - Q(s,a)]$
9:      $s \leftarrow s'$; $a \leftarrow a'$;
10:   until s is terminal
11: until maximum episode
```

2.4 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning is an area of Reinforcement Learning that allows the division of a problem into sub-tasks, making it simpler to solve.

The normal Reinforcement Learning has a set of problems in complex real world domains such as:

- Curse of dimensionality due to the great number of states and actions increasing the learning time.
- Generalization problem - if the observation space is continuous, the agent cannot visit every state to find the optimal actions for those states.
- Exploration vs Exploitation problem - The agent needs to choose whether it is a moment of explore to new states or to choose the best action to lead him to the objective. The balance between these two is a challenge for tasks with a high number of states and actions where the computational time is a factor at stake.

Hierarchical Reinforcement Learning can be used to beat these challenges, deconstructing a task into sub-tasks easier to solve. This method has many advantages like:

- Policies learned in each one of the sub-tasks can be reused.
- Value functions can be shared between sub-tasks, accelerating the learning process of a new sub-task.
- State abstraction can be applied, allowing a system to ignore irrelevant details to the task reducing complexity, time and resources spent on learning.

Most of the algorithms used in the Hierarchical Reinforcement Learning field are based on Semi-Markov Decision Processes (SMDP). The SMDP is a variant of the Markov Decision Process where the abstract actions can take random time-steps to complete, with another variable being necessary to terminate this abstract action. SMDPs can model continuous-time events and discrete-time, usually treating the system as remaining in each state for a random waiting time transitioning instantaneously to the next state at the end.

A simple type of abstraction is a “macro”, which is a sequence of actions that can be called upon as if it were a simple action. For example a “macro” can be “Turn right at the door” and this “macro” have a subroutine where the actions to solve this “macro” are maintained. A “macro” is an open-loop control policy, which can be generalized as a partial closed-loop policy where it is necessary to define a subset of the state set and have a termination condition. This partial policy is called options [24] and with this in mind the wait time in each one of the options becomes only its duration instead of a random number.

There are a few known approaches in Hierarchical Reinforcement Learning that will be presented ahead: Options, HAM and MAXQ.

2.4.1 Options

Options is a framework that generalizes the primitive actions to include temporally extended courses of actions, “macros”.

It is a triple composed by:

- A policy $\pi : S \times A \rightarrow [0,1]$.
- A termination condition β that shows the probability of the option termination in the given state, $\beta : S^{+} \rightarrow [0,1]$.
- And an initiation set, $I \subseteq S$.

An option is available in a state if that state belongs to the initiation set and follows the policy π until it reaches the termination point. If the agent is in a state s belonging to I, the action taken has a probability of $\pi(s, a)$ and in the next state s' the option could terminate with probability β or continue. When an option ends, a new option can be selected by the agent.

In [25], a new algorithm based on this Options framework is presented where the option is not necessarily taken until the end of it because that may constrain the agent. The objective is to know if at the state s_t, arrived at by choosing the option o, the value $Q(s_t,o)$ to continue with o is less than the value obtained after interrupting the option o and continue with a new option. This method is applied in a mobile robot in an unknown environment to resolve motion planning tasks. The robot has three options: move-to-goal, avoid obstacles and follow walls and it uses the Q-Learning algorithm in the learning process. The results of this project show that the new algorithm finds a solution faster than the normal Options framework in simple environments, however, the normal Options framework supports the definition of flexible policies giving the capacity of better adaptation to unknown environments.

2.4.2 HAM

Parr and Russell, in 1998, developed an hierarchical approach called *Hierarchies of Abstract Machines (HAM)*. This technique uses the SMDP concept, but simplifies complex MDPs restricting the set of policies instead of increasing the set of actions. HAMs consist of non-deterministic finite state machines where the transition between states invoke lower-level machines. This nondeterminism is represented by choice states where the optimal action is yet to be decided or learned. To solve a problem, there can be no constrains at all but also a full specified solution. However, a intermediate solution is normally used, leaving to the learning algorithm the discovery of the lower-level option to take at each time [26].

The machines for HAM are defined by:

- A set of states that are composed by four types:
  - **Action** states, which execute an action in the environment.
  - **Call** states, which run another machine as a subroutine.
  - **Choice** states, which select the next state machine nondeterministically.
  - **Stop** states, which end the execution of the machine and return the control to the previous call state.
- A transition function that determines the next machine state.
- A start function that determines the initial state.

HAMs can reduce the time needed to learn a new environment, focusing on the exploration of the state space using constraints. In [26] the authors created a variation of the model-free algorithm Q-Learning called HAMQ-Learning that learned directly in the reduced state space without any model transformation and compared the two of them, showing in the results that the new algorithm learns much faster (approximately thirty times faster) than the traditional one. These results show that constraining the set of policies considered for a MDP is possible using the HAMs, accelerating the learning process and providing a transfer learning method for other tasks.

2.4.3 MAXQ

MAXQ allows to decompose a main task into sub-tasks that can also be decomposed into sub-tasks to a point where a sub-task is only composed by primitive actions as stated by T. Dietterich in [27]. Having the task hierarchy, the goal of the MAXQ is find recursively a optimal policy. In order to do this, the author created an algorithm called Hierarchical Semi-Markov Q-Learning (HSMQ) that is applied to each of the tasks simultaneously to learn the policies. Each sub-task **p** has its own Q function Q(p,s,a) that describes the expected total reward of choosing the sub-task **p** in an initial state **s** while executing the action **a**. MAXQ allows to decompose value functions that HSMQ can not do.

MAXQ decomposes Q(p,s,a) into a sum of two components:

- V(a,s) - expected one-step reward upon choosing action **a** in state **s**.
- C(p,s,a) - expected total reward upon completing the task **p** after **a** has returned.

$$Q^{\pi}(p, s, a) = V^{\pi}(a, s) + C^{\pi}(p, s, a) \tag{2.15}$$

If this method is applied recursively, it decomposes the Q function of the root task into a sum of the Q values of its sub-tasks representing the value function of a hierarchical policy. This method provides the opportunity to exploit state abstraction, ignoring large parts of the state space.

In [28], T. Dietterich evaluates the Algorithm MAXQ-Q with and without state abstraction against the traditional Q-Learning showing that it converges much faster to an optimal policy. It shows that MAXQ support the reuse and sharing of a sub-task because of the decomposition of the value function into value function for each one of the sub-tasks.

In [29], the authors created an algorithm that combines two of the frameworks already presented: Options and MAXQ. The MAXQ framework introduces knowledge to the Reinforcement Learning and the Options framework creates automatically the hierarchies in a problem of two robots collecting trash in a partially known environment. Being a multi-agent task becomes more challenging, because if both robots are treated as a single agent, the number of the state space and action space increases exponentially (curse of dimensionality), and if one of the agents is treated as part of the environment, it makes the environment non-stationary and non-Markovian. The method proposed can solve the whole problem dividing the task into sub-tasks and decomposing the value function faster than the traditional MAXQ.

2.5 Reinforcement Learning for Mobile Robotics

This section presents some selected examples to solve mobile robotic problem using Reinforcement Learning.

In [30] the main objective is to show the use of Q-Learning for a navigation problem in an unknown environment, calculating the shortest path to the goal state and avoiding obstacles through the analysis of captured images that are processed using a canny edge detector. Capturing images from the environment allows the creation of a grid map of it and the cells of this grid are the state space of the algorithm. Taking this into account, this problem can be solved as the Grid world exercise showing that the Q-Learning algorithm can be used to solve real world navigation problems.

Q-Learning was again used in [31] to help a mobile robot to leave an unknown maze. This maze has a spiral shape and the robot is placed right in the center of it, which has to leave the maze only with the help of sonar sensors and the Reinforcement Learning method. The robot only has three possible actions and in the state space, each state is defined by a vector representing the left, right and front distances and their relative amplitude. These states are then divided into two classes: health and sub-health states. The health states show that the obstacles are very far away from the robot and the sub-health states indicate that at least one obstacle is near the robot. The reward function is made with the help of the user, who gives the robot information about the best action to choose in each of the states. In the end, the robot learned how to solve the maze with the help of the Q-Learning method converging to a fine solution after 300 steps.

In [32] the reader is presented with a Reinforcement Learning approach to allow a service robot, Roomba, that has a topological map of the environment, to learn a path from one location to another. The environment is a set of rooms and in each of them the robot only has a set of different movements to make, given by the user as a navigation map. The algorithm used was the Q-Learning and in order to traverse through each room towards its objective, the robot uses the A* search algorithm to find the shortest path. The results obtained were positive, showing that the Q-Learning is a viable method to use in mobile robot navigation.

In [33] the authors applied an improved Q-Learning algorithm in a two-dimensional setup where a robot tries to learn its path without colliding with any obstacles or preys to a final destination. The algorithm differs to the traditional one because all the state-action pairs are stored and replayed backwards to propagate the refined Q values from any state to the goal. Each of the states has information about the x and y coordinates of the robot and if the robot sees an obstacle or prey in front, right, left or rear of it. This method proved to be better than the traditional Q-Learning due to the higher rate of change of the Q values accelerating the convergence of the algorithm.

Model-based methods are increasingly used in the field of mobile robotics, whether in obstacle avoidance, parking, overtaking in transit, etc., with more and more research on new or more efficient algorithms.

In [34] and [35] it is proposed a model-based algorithm based on a cell mapping technique, an extension to the *CACM* technique, for a motion planning done online in nonholonomic mobile robots. These robots have the peculiarity of not being able to change their pose instantly to any of the available directions. The CACM technique is based in the Bellmans principle of optimality for continuous dynamic systems and on cell-to-cell mapping techniques discretizing the state variables of the system, dividing the state space into cells. This method uses the Adjoining Cell Mapping (ACM) technique, whose main objective is the creation of a cell mapping where only trasitions between neighbour cells are allowed. A comparison between this algorithm and Q-Learning is made, in which case the algorithm used obtains better results.

In [36] a new algorithm based on a model-based approach using the CACM technique is presented, which can be directly used in a real mobile robot with integrated vision system and whose mission is to perform a docking task without requiring a simulated environment. This algorithm is compared to a linear controller improving the results and speeding up the docking process, having approximately 60% of the cells controlled against only around 40% using the linear controller method.

In [37] the reader is presented with a quasi-online Reinforcement Learning approach where the robot while exploring the environment also creates a probabilistic model of the same being, compared later this algorithm with the more classic methods like Q-Learning, Dyna-Q and Prioritized Sweeping. The algorithm created is based on Prioritized Sweeping with directed exploration and a transformed reward function. Exploration is a very important factor in Reinforcement Learning, and the Directed exploration, unlike the undirected one, is way more efficient because explores unevenly around the current policy, allowing to converge to a more complete model. The transformed reward function allows more information to be given to the robot during the learning process, designing a straightforward reward function and afterwards transforming the original reward function to accelerate the learning process. This approach shows superior results than any of the traditional algorithms converging quickly to optimal solution.

In [38] the authors describe a variant of the Rmax algorithm used for planning on a relocatable action model, (RAM-Rmax). Rmax is a powerful model-based algorithm with a near-optimal performance thanks to its polynomial computational complexity [39]. The Rmax is divided in two steps: in the first one, the agent is in the computational stage, using its knowledge applying the optimal policy until the end of the episode or the visit of a new state, and in the second step, the agent updates the rewards and transition probabilities of the model for each of the possible actions, recalculating the optimal policy. The results obtained using a Grid World showed that the RAM-Rmax was faster than the traditional Rmax and Q-Learning to begin to follow the optimal policy, and finally, the authors compared the algorithm created with the Rmax in a real environment and the agent almost immediately followed the optimal policy using the RAM-Rmax algorithm.

Chapter 3

Experimental Setup

One of the objectives of this dissertation was the development of a robotic system able to perform navigation experiments in a real maze-like environment. This chapter describes the experimental setup used throughout the work from the initial steps to the final solution, including the overall system architecture at the hardware and software levels.

3.1 Overall System's Architecture

One of the objectives of this dissertation was the development of a robotic system able to perform navigation experiments in a real maze-like environment. This chapter describes the experimental setup used throughout the work from the initial steps to the final solution, including the overall system architecture at the hardware and software levels.

Fig. 3.1 illustrates the devised robotic systems architecture. The communication between the host computer and the remote mobile robot is established through a wireless system, such that high-level commands are sent from the host computer to the mobile robot and sensorial measurements (feedback) flow in the opposite direction. In what concerns the software development, the first decision was to adopt the ROS middleware for the robotic project since it provides multi-platform support, distributed programming, real-time execution, more and more compatible products and powerful simulation tools.

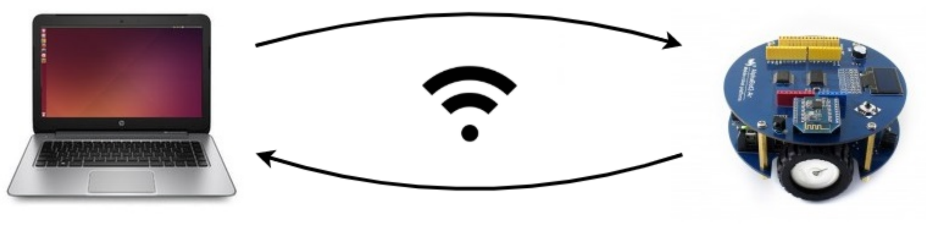


Figure 3.1: Main computer vs Arduino diagram.

The selection of the robotic platform and the mobile robot simulator required further study having in mind several requirements. On the one hand, the basic requirements for the mobile robots were as follows:

1. As cheap as possible to overcome budget limitations;

2. Small-size to operate in maze layouts whose corridors are about 30 cm width;

3. Self-contained in terms of energy supply and long battery life since the robot may have to operate during long periods of experimentation;

4. Support wireless communications to increase the range of applications;

5. Equipped with low-level sensing/actuation systems.

Multiple sensors, working on different principles, can be installed on a mobile robot so that it can perceive its environment. However, in the context of this work, the robot platform should rely, above all, on a set of infrared sensors distributed around the perimeter of the robot to build a local map of a maze-like environment. The presence of sonar-sensing could provide complementary information useful, mainly, during the training phase.

On the other hand, a simulator has an important role when applying Reinforcement Learning in the context of robotics. On the one side, experience on real physical robots is, usually, expensive, tedious to obtain and hard to reproduce. For example, model-free methods can learn in real-time, but often need thousands of interactions with the environment to learn a good policy. On the other side, there are limitations to using simulated robots: experiences performed in simulation environments may be hard to transfer to the real-world since modeling errors and under-modeling can lead to quite different behaviors (unpredictable behaviors).

In line with this, the main concern was to find the right hardware and software tools to implement a final solution without spending an excessive amount of time with engineering solutions that, often, end up in reinventing the wheel. Therefore, the following subsections review both mobile platforms and robotic simulators available in the market according the above requirements.

3.1.1 Mobile Robot Platform

Recent advances in sensor miniaturization and computational power has expanded the offer of small, low-cost mobile robots. This subsection briefly describes the three small-size robots under analysis: ct-bot, Khepera IV and Alphabot2 (Fig. 3.2).

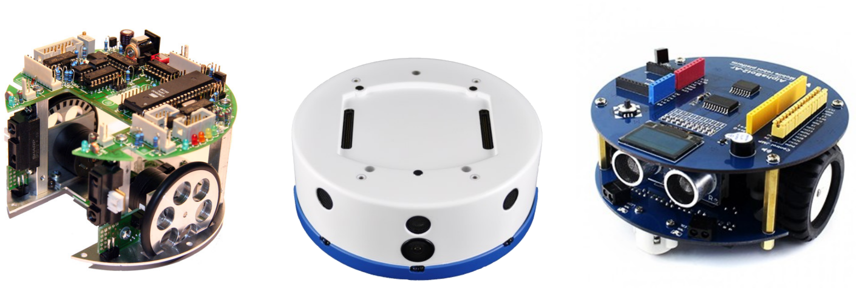


Figure 3.2: Example of mobile robot platforms under analysis: ct-Bot (left), Khepera IV (middle) and Alphabot2 (right).

c't-Bot is made in Germany and offered by eMedia and Segor electronics [40]. It was made to avoid obstacles and compete with other c't-Bots solving mazes. Due to a wide variety of sensors to measure distance, light, lines, abysses, flaps, it is a very viable robot for mobile robot navigation activities. It has a peculiar particularity since all its components are Through-Hole Technology (THT), unlike the most known cases that use Surface-mount Technology (SMD) components. It has a similar structure to the AlphaBot2, with a small size to pass through very narrow spaces, it has sensors, actuators and a battery at the base while the electronics are on the top plate. It also has a rectangular opening at the front so one can push small objects. It contains an Atmel ATmega32 as a microcontroller that has the mission to coordinate the movement, read the values of the sensors and communicate with the outside. Along with the robot, an open source simulator was created, c't Sim, used to reproduce the real environment and to test the bots without having the problems of the physical environment and incoherent measures.

Khepera IV is the latest reference from the KTeam company, headquartered in Lausanne, Switzerland, which began as a startup created at the Ecole Polytechnique Fédérale de Lausanne (EPFL) [41] . This company is dedicated to the creation of small robots that contains state of the art features to carry out experiments and studies in mobile navigation, real-time programming, artificial intelligence and multi-agent systems. With a Linux core with Bluetooth, Wi-Fi, accelerometer, gyroscope, it is a robot with a high modularity, with capacity to add a varied range of extensions. It has an array of infrared sensors for obstacle detection and line following and ultrasonic sensors to detect objects at greater distances. In addition to infrared and ultrasonic sensors, the Khepera IV contains an integrated camera capable of color filming at a rate of 30 fps. It would be the ideal robot to perform all the tests done in this work, but due to the excellent quality of electronic components, it is an extremely expensive robot, costing around two thousand and three hundred euros.

AlphaBot2 is a second-generation robot built by a company named Waveshare and based in Hong Kong, China [42]. The robot has three variations, each for a distinct board type: one compatible with Arduino, the second compatible with Raspberry Pi Zero W and the last one with Raspberry Pi 3 Model B. It is an interesting robot because in spite of the low price it has systems of line tracking and obstacle avoiding already integrated.

The final choice fell on the Alphabot2 robot compatible with Arduino. In addition to a lower cost, this is an Arduino solution that presents an easy-to-learn programming language (derived from C++) and provides open-source tools. The version chosen was the version compatible with the Arduino board, specifically the Arduino Uno Plus, and it is interesting to know all of its components: (see Fig. A.1 and Fig. A.2 [43])

Arduino Uno Plus

The company Waveshare has created an Arduino board compatible with the Arduino Uno R3, being an improved version with the possibility of supporting more types of shields (Fig. 3.3). The table 3.1 shows some of the differences between these two Arduino boards [44].

	Arduino Uno Plus	**Arduino Uno R3**
Operating Voltage	5V/3.3V	5V
USB Connector	Micro USB	USB Type B
Oscillator	Crystal Oscillator	Ceramic Resonator
ADC Channels	8	6

Table 3.1: Comparison between Arduino Uno Plus and Arduino Uno R3.

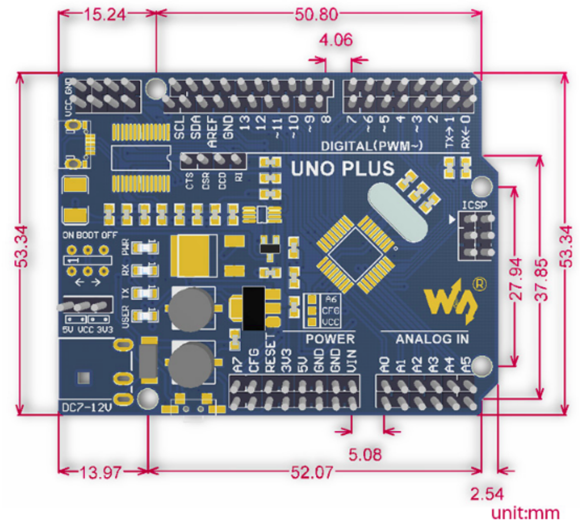


Figure 3.3: Arduino Uno Plus dimensions.

The dual operating voltage is necessary to support a wide variety of shields. The Micro USB connector is used to allow a better fit by not blocking the placement of shields. The crystal oscillator provides more accurate clock measurements than the ceramic resonator. The CFG and Reserved pins can be configured as analog pins, however, by default, the CFG pin is connected directly to the VCC through a resistance of value equal to 0, this connection had to be desoldered in order to be able to use the pin as an analog port.

Sensors

- **ITR20001/T - Opto Interrupter** - The ITR20001/T infrared sensor presented in Fig. 3.4, is commonly used to follow lines, but in this particular case it will be used to detect and decode bar codes in the maze. The Alphabot has five of these sensors in order to solve pathfinder exercises, but in this work, only one of them will be used, because there is no need to receive the same information from the five sensors simultaneously. The ITR20001/T has an infrared emitting diode to a NPN silicon phototransistor, side by side, which allow the detection of the robot passage over a black and white bar code and distinguish both colors, as well as the base of the maze.In order for it to function properly, in the moment the robot starts up, it is necessary to calibrate these sensors by moving the robot over one of the bar codes [45].

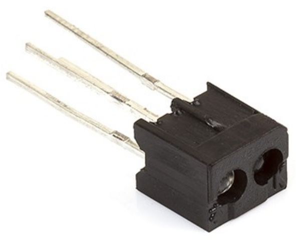


Figure 3.4: Reflective infrared photoelectric sensor, ITR20001.

- **HC-SR04 - Ultrasonic module** - The HC-SR04, presented in Fig. 3.5, is a distance sensor composed of a transmitter and receiver, with a capacity of measuring distances between 2cm and 4m, with an accuracy of approximately 3mm. This sensor emits ultrasonic signals that reflect in the object to be reached and return to the sensor which allows to calculate the distance to the target taking into account the speed of the signal [46].

  The speed of the ultrasonic signal emitted by the HC-SR04 Sensor corresponds to the sound velocity, which is approximately 340 m / s, so if the sensor is at a distance x from the object, the signal will travel a distance equivalent to 2x , the wave is sent by the sensor and rebounded in the obstacle, then it travels 2 times the requested distance.

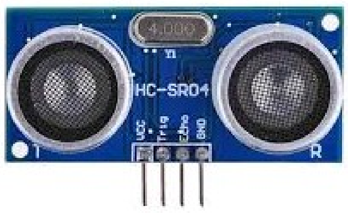


Figure 3.5: Ultrasonic module, HC-SR04.

Actuators

- Omni-direction wheels.
- Dual H-bridge motor drive,TB6612FNG.
- N20 micro gear motor reduction rate 1:30 6V/600RPM.
- Battery holder, for 14500 batteries.

3.1.2 Robotic Simulators

A review of the literature on mobile robot navigation based on RL techniques shows the usefulness of realistic simulators for testing and evaluating performance of the algorithms in a cost-effective and timely way. [47, 48]. These programming tools involve the proper selection of a physical engine, as well as the realistic simulation of all robots functions and environment interactions, including the dynamic motion of the robot, control modules and sensor characteristics. Among the most used simulators are V-REP, ARGoS, Gazebo and MATLAB/Simulink, whose most important features are described below.

Virtual Robot Experimentation Platform (V-REP) is a robot simulator made by Coppelia Robotics used for fast algorithm development, simulation, prototyping and verification in cases of remote and safety monitoring, hardware control and factory automation simulation.[49]. V-REP is a very complex simulator that spends too many resources. It has the advantage of offering several physics engines, the possibility of interaction with the user during a simulation and allows mesh manipulation. It is a very effective simulator when the number of robots is low.

ARGoS is a robot simulator that was initially developed whitin the Swarmanoid project, funded by the European Commission and that started in 2006. This project had as its main objective the design, implementation and control of an "army" of autonomous robots of three types: *eye-bots*, *hand-bots*, *foot-bots*[50]. Later, it became the main simulator tool in some European projects like ASCENS, H2SWARM, E-SWARM and Swarmix. Through these projects it is possible to realize that it is a simulator made for a higher set of robots simultaneously, having a high efficiency in many applications, such as exploration of hostile environments, nano-scale medicine and disaster recovery.[51]

ARGoS is a simulator that privileges performance to the properties of the robot, the environment and the physical characteristics. It has the disadvantage of not being able to import 3D meshes, and as a consequence that it would be an arduous task to construct the model of the robot Alphabot2-AR. Also, it would not be the most suitable simulator to carry out the tests of this dissertation because in them only one robot will be used.

Gazebo is a robot simulator that began to be developed in 2002 at the University of Southern California [52]. Its creation is due to the fact that a simulated environment is necessary to mimic the behavior of a robot in a real environment. Over the years, the software has been improved until the Open Source Robotics Foundation (OSRF) entered the scene in 2012, and since that date, the Gazebo has been growing a lot and it is already in version Gazebo 10.0. Unlike ARGoS, the Gazebo has a high number of features almost at the level of V-REP, but has a much simpler interface and a large number of robots already modeled. However, Alphabot2-AR is not one of them and would have to be created. Although it allows the importation of 3D meshes, it is not possible to edit them. The biggest problem with the Gazebo simulator is the number of interface-based problems and installation of dependencies.

Most of the work focuses on optimizing the performance of RL algorithms from simulation models (prior knowledge) before transferring the best solution on the physical system, for example, in the form of an initial policy. It is often observed that solutions optimized in simulation are inefficient on the real robot due to modelling errors and under-modelling effects. Solving the virtual versus real gap requires often optimizing the parameters of the solution on the real robot. In this work, it was decided to start by using a kinematics simulator to be developed in MATLAB (described in Subsection 3.3.4), focusing on the simulation of the IR-sensors interacting with the environment. The initial hypothesis is that, in the context of a HRL approach, a simple kinematic model provides a chance to learn a non-optimal policy, but able to successfully achieve the desired goal without a clear reality gap.

3.2 RLAN-bot Final Prototype

The final mobile platform, hereinafter called RLAN-bot(Reinforcement Learning Autonomous Navigation Robot), resulted from the integration of a set of components missing in the Alphabot2, including a wireless communications module and a belt of infrared sensors distributed around the perimeter of the robot:

Wireless Communication Module - To enable communication between the robot and the computer, a wireless communication module, ESP8266 ESP-01, is used (Fig. 3.6). This module allows the Arduino installed in the robot to access the Wi-Fi network in which the computer is also connected. Instead of being connected to the Arduino, it could act alone as it is a System On Chip (SOC), not needing a microcontroller, having two General Purpose Input/Output (GPIO)s. However, as two GPIOs are not enough for this project, the module is only used to make wireless communication.

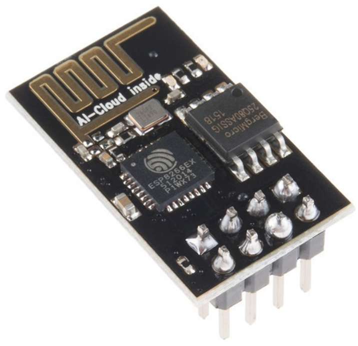


Figure 3.6: ESP8266 ESP-01 module.

This module supports 802.11 b/g/n networks which are the most common networks nowadays and can be used as Access Point, Station, or both simultaneously, allowing the sending and receiving of data. ESP8266 ESP-01 comes with pre-installed firmware with AT commands. These commands enable quick connection to the Wi-Fi network and are compatible with the Arduino IDE. The connection between the module and the Arduino is made through an adapter showed in Fig. 3.7, that has a voltage regulator that converts the voltage sent by the Arduino, 5V, into the supply voltage that the ESP8266 accepts, 3.3V.

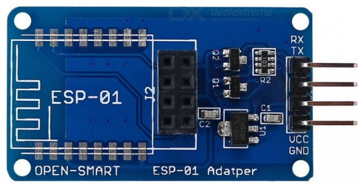


Figure 3.7: ESP-01 adapter.

The communication between both Arduino and ESP8266 is made using the RX and TX pins, Table. 3.2. The module supports Transmission Control Protocol/Internet Protocol (TCP/IP) and User Datagram Protocol (UDP) as communication protocol.

Arduino Uno Plus	**ESP-01 adapter**
5V	VCC
GND	GND
D9	RX
D8	TX

Table 3.2: Connection setup between Arduino and ESP-01 adapter.

Analog Digital Converter ADS1015 12 Bit - In order to be able to add more sensors to the robot, it was necessary to add a module that would increase the range of analog pins.

An analog-to-digital converter was used, ADS1015, with a high precision of 12 bits, whose communication with the Arduino is made through I2C (Fig. 3.8).

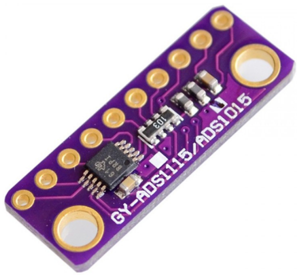


Figure 3.8: Analog Digital Converter ADS1015 12 Bit.

This module has a very low power consumption, in the order of 150 μA, an extremely fast data conversion rate from 128 SPS (samples per second) to 3300 SPS and has a multiplexer internally that allows to have two differential or four single-ended input measurements. It also has a digital comparator for under and overvoltage detections.

Its connection to the Arduino is quite simple since it is done through I2C, using the addresses available between 0x48 and 0x4B. The analog pins connect to the output voltage pins of each of the four infrared sensors that will be discussed below. The entire connection scheme can be seen in Table 3.3.

ADS1015 Module	**Arduino Uno Plus**	**Infrared Sensors**
VCC	5V	VCC
GND	GND	GND
ADDR	GND	-
ALRT	-	-
SDA	SDA	-
SCL	SCL	-
A0	-	Vo (sensor 1)
A1	-	Vo (sensor 2)
A2	-	Vo (sensor 3)
A3	-	Vo (sensor 4)

Table 3.3: Connection setup between Arduino and ADS1015 Module and Infrared Sensors.

Sharp GP2Y0A41SK0F - As the robot used does not have distance sensors for all the needed directions, only having an ultrasonic sensor turned forward, it required a set of sensors that accurately measures distances in the six directions (left, front left, right, front right, front and rear). For this purpose, six infrared sensors, Sharp GP2Y0A41SK0F were used [53] (Fig. 3.9). These sensors have the basic operation of an infrared sensor: it has a Position Sensitive Detector (PSD) , an Infrared Light Emitting Diode (IR-LED) and a signal processing circuit. As a result of its triangulation method in distance measurement, it is not much influenced by external factors such as ambient temperature and reflectivity of objects. It can measure distances that range from 4 cm to 30 cm. Thus, in order to allow the robot to be able to have distance values very close to its ends, these sensors were placed in the center of the robot in its upper part, fixed in a support made of Nylon that accents on the Printed Circuit Board (PCB) designed to make all the connections to the robot (Fig. B.1).

Figure 3.9: Sharp GP2Y0A41SK0F.

Each sensor has three pins: VCC, GND and Vo, whose connection is necessary to make them work. The required supply voltage ranges from 4.5V to 5.5V so it is connected to the 5V pin of the Arduino, the GND is connected to the GND of the Arduino, and the output voltage of four of them is connected to an analog pin of the ADS1015, and the other two sensors are connected to the Arduino pins CFG and Reserved. The distance values are calculated by the Arduino. As it can be seen, the measured distance is obtained through a voltage value having a curve similar to that of Fig. 3.11.

The integration of the hardware described above required the development of a PCB that could serve as a shield to connect these components to the Arduino. Initially, it was thought that four sensors (front, back, left and right) would be sufficient for the learning process and a PCB was created only with the connection for these four sensors. A pole was also designed to properly install the four sensors that would be attached to the top of the board. This parallelepiped pole had a base measuring 14mm of side and a height of 60mm. Latter, with the first experiments, there was a lot of ambiguity in certain states that could not have the same action. Therefore, a new PCB was designed with the connections for two more infrared sensors, one diagonally to the left and one diagonally to the right. In order to incorporate these new sensors it was necessary to construct a new pole to support all six sensors at the top of the robot. This pole has a parallelepipedic base with 14 mm wide and 15 mm high, and above it, a hexagonal prism with five sides with 21 mm where the front sensors are fixed, left and right, and a side with 52 mm where the rear sensor is fixed. This prism is 45 mm high so the pole has a total height of 60 mm. The RLAN-bot is shown in Fig. 3.10.

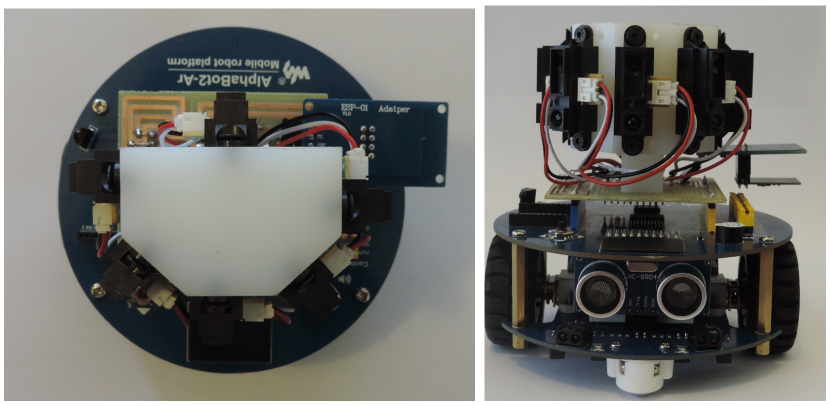


Figure 3.10: Final version of the robot.

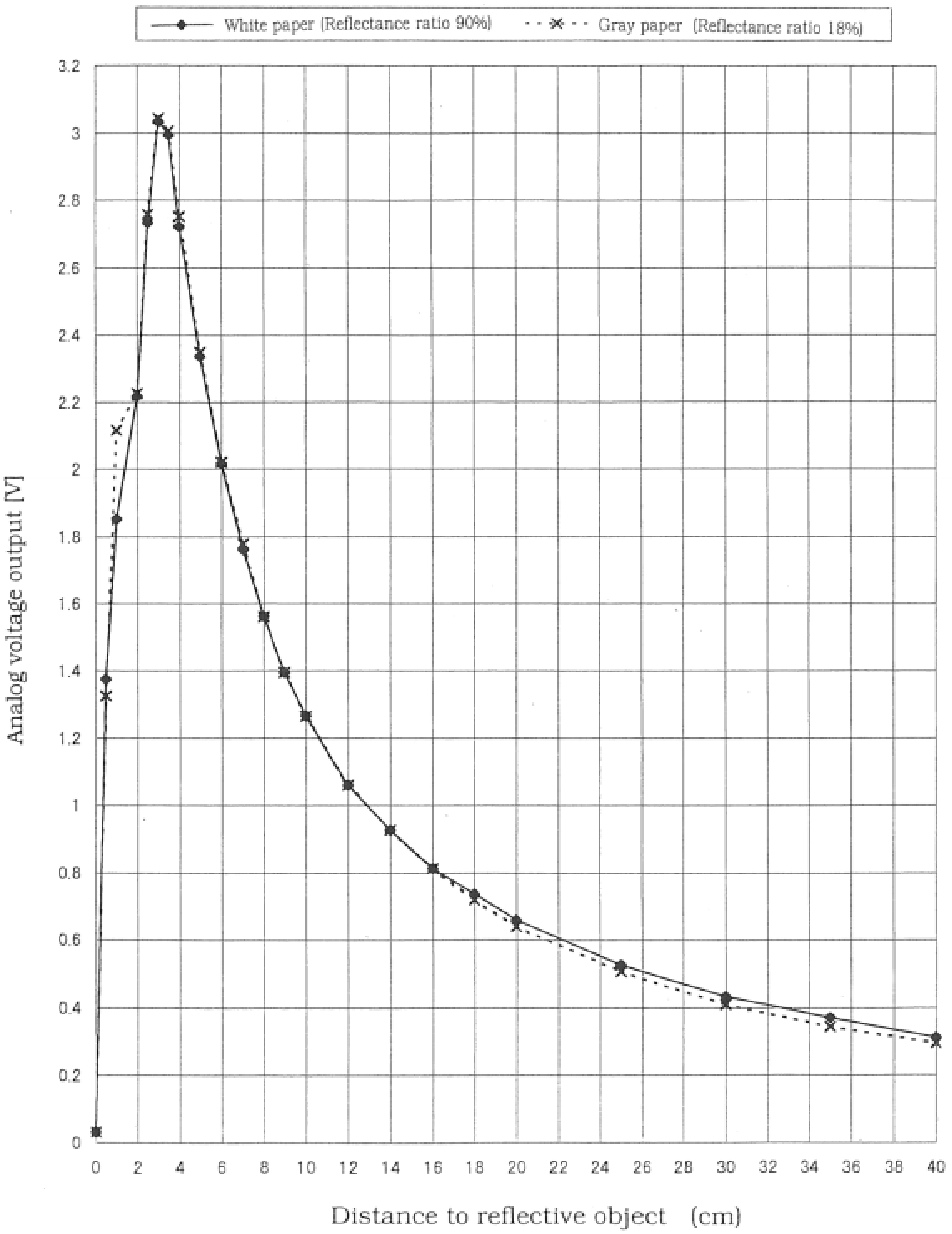


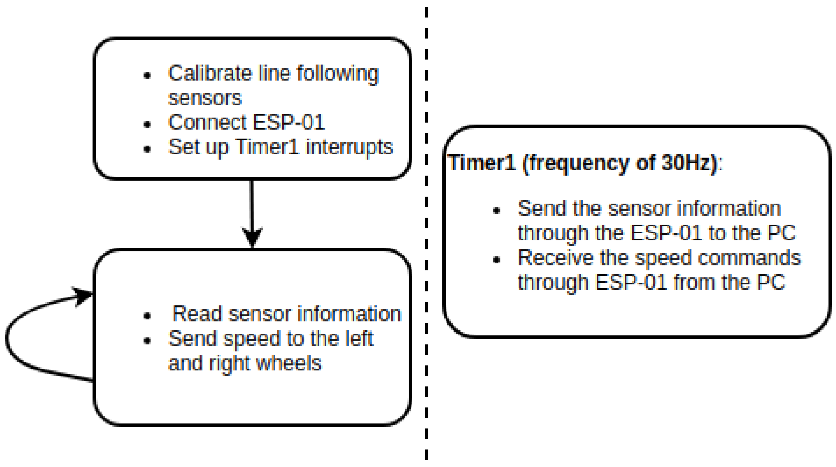
Figure 3.11: Output voltage corresponding to the distance.

3.3 Software Tools

This section identifies all the software tools either to perform the experiments with the physical system or to simulate the operation of the robot, including the Arduino IDE, the ROS development environment and the mobile robot simulator developed in the context of this work.

3.3.1 Arduino IDE

Arduino IDE is an application used to write and upload programs on Arduino. It supports programming languages like C and C ++ and contains a large number of libraries to facilitate contact with the user. This software is supported by the various existing platforms, Windows, Linux and Mac. This application was used to write and upload in the Arduino Uno Plus the program that makes the reading and calculation of the distance of the infrared sensors, the reading of the infrared sensors of line tracking, the configuration of the Wi-Fi module and the reception and sending of information to and from the computer (Fig. 3.12).



• Calibrate line following sensors
• Connect ESP-01
• Set up Timer1 interrupts
• Read sensor information
• Send speed to the left and right wheels
Timer1 (frequency of 30Hz):
• Send the sensor information through the ESP-01 to the PC
• Receive the speed commands through ESP-01 from the PC

Figure 3.12: Flowchart diagram of the program running on the Arduino.

3.3.2 ROS and Development Environment

Robot Operating System (ROS) is an open-source middleware that facilitates the communication between the computer operating system and the external equipment like sensors and robots. It is easy to implement in Python and C++ and commonly used by roboticists for being a thin and easy testing middleware and having a dedicated library with clean interfaces. There are a few versions of ROS already developed like the ROS-Industrial dedicated to robots for manufacturing. Others are still in development, like ROS-M for military companies, H-ROS for interoperable components and ROS 2.0, an upgraded version of ROS with the latest technology.

ROS communicate between ROS nodes that represent the executable code, a process. ROS nodes do not need to be all exclusively on the same computer, making it a very flexible system. They can be on the computer or spread between computer and robot. These nodes communicate through messages, each one organized into categories called topics. Nodes can collect information from the topics, **subscribe**, or **publish** information on them. There is a Master Node that provides naming and registration services to the remaining nodes, tracking subscribers and publishers to the topics and providing the communication between nodes. The communication protocol used between the nodes is the TCP/IP (Fig. 3.13).

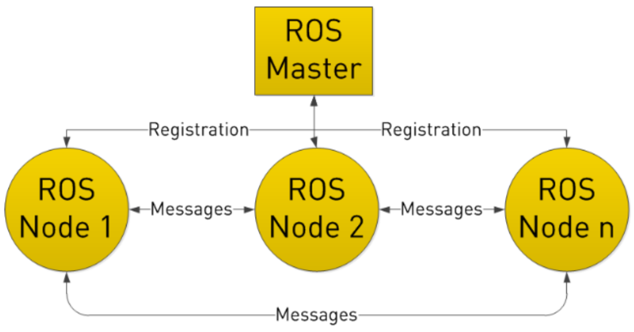


Figure 3.13: Basic ROS diagram.

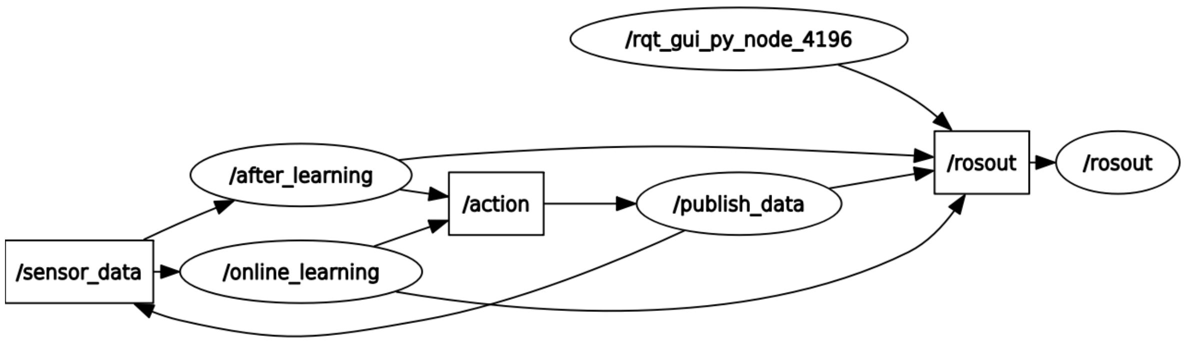


Figure 3.14: ROS architecture graph.

For the incorporation of the ROS in the project, a tool is needed to write all the C ++ code of the reception of the information of the sensors and consequent analysis and process. The Visual Studio Code was used to edit the created code and to save the entire procedure directly to GitHub.

The ROS architecture allows to create nodes that function as processes, each of the nodes having a certain function being able to communicate between them through the topics. In this project, three distinct nodes were used as can be seen in Fig. 3.14:

- *publish_data* - This node allows to receive the sensory information sent by the robot remotely with the help of the Wi-Fi module, it analyzes if the robot has changed sub-task taking into account the information of the checkpoints, and later publishes the values given by the sensors and the checkpoint number in the dedicated topic to save this information, *sensor_data*. Subsequently subscribes to the topic *action* to be able to send to the robot the command of movement to use every moment.

- *online_learning* - This node is responsible for performing the learning algorithm Q-Learning in each of the sub-tasks with or without the learning tables previously obtained in simulation. For this, it subscribes to the topic *sensor_data* to calculate the state in which it is at each moment and publishes in the topic *action* the action that the robot has to perform.

- *after_learning* - This node is in charge of obtaining the maximum Q value in each state that the robot is, taking into account the learning tables obtained in simulation. For this, it subscribes to the topic *sensor_data* to calculate the state in which it is at each moment and publishes in the topic *action* the action that the robot has to perform.

3.3.3 Communication Protocols

In order to configure the communication between the robot and the computer, it it first necessary to understand the operation of the AT commands and the ESP-01 module. This module version has a baudrate of 115200, so the first thing to do is initialize the communication with this baudrate. To configure the module, it is necessary to use an existing library in Arduino: SoftwareSerial. This happens because the ESP-01 needs to create a Serial connection apart from that that already exists in the Arduino. This library does not work correctly with the default baudrate that the ESP-01 brings from source, 115200, so it is necessary to lower this baudrate to 38400. To do this, the existing firmware in the Wi-Fi module has a specific command:

- AT+UART_DEF= <baudrate>, <databits>, <stopbits>, <parity>, <flow control>.

Having the module with the correct baudrate, it can be configured as a Station. To setup the module, it is necessary to follow a few steps:

- AT+RST - Restarts the module.

- AT+CWMODE= <mode> - Set the Wi-Fi Mode as Station, Access Point or both. In this case, the chosen Wi-Fi Mode is the Station one.

    1. Station;
    2. Access Point;
    3. Both.

- AT+CWJAP="ssid of the network","password" - Connects the Wi-Fi module to an Access Point.

- AT+CIPMUX= <mode> - Enable or disable multiple connections. In this case, it is used the single connection.

    - 0 - Single Connection.
    - 1 - Multiple connections.

- AT+CIPSTART= “type of connection”,”remote IP address”, <remote port number>, <detection time interval (TCP)> - Establishes one of the three connections: TCP, UDP or Secure Sockets Layer (SSL). In this case, the UDP protocol is used because it is faster than the TCP/IP protocol due to non-retransmission of loss packets.

After the configuration is made, the robot is ready to send the values of the sensors and receive the commands to move. In order for this communication to be possible, there are two more commands that will be used in loop since it will be a continuous activity:

- +IPD,<data length>:<data> - Receives network data. After this command, it is used the *read* function of the SoftwareSerial library that returns a character that was received on the RX pin of the Arduino.

- AT+CIPSEND=<data length> - Sends data of designated length. After this command, it is used the *write* function of the SoftwareSerial library that writes binary data to the serial port.

Initially a frequency of work of 50Hz was tested, but the command +IPD did not allow it because many times it did not respond correctly, thus this frequency was lowered to a value that gives good results, having fixed the frequency in 30Hz, both in the Arduino and the main computer. To keep sending and receiving information at a fixed rate, a timer was used in Arduino and the same rate was designated in the ROS environment.

3.3.4 Mobile Robot Simulation in MATLAB

MATLAB is a widely used software in the field of Machine Learning, robotics, computer vision and image processing. Its matrix-based programming language greatly simplifies the mathematical operations being of a high computational speed. Due to the high number of already existing functions, it is an essential tool in the creation of simulators for a case study such as the one studied in this work. MATLAB is a tool that greatly facilitates the analysis of data and the creation of plots.

During the dissertation, MATLAB was used in the first phase to perform an exercise to familiarize with Reinforcement Learning, solving a Gridworld environment with an initial state and with the intention of discovering the path to the final state, being able to use only four types of drive: go ahead, go back, go right and go left. For this exercise, the Q-Learning algorithm was used.

For the simulation of the real environment, it was decided not to use a simulator as rich as the three already mentioned, but to use MATLAB instead.

The functions created were made parametrically so as to create different environments, to change the dimensions of the robot, to change the number of sensors and the maximum and minimum distances that these measure, and to change the distances for the different levels of the sensors. Altogether there are seven functions, each with a specific purpose:

Fig. 3.15 illustrates the interface created in MATLAB for simulating the navigation of a mobile robot in a maze-like environment. For that purpose, a library of MATLAB-functions was created for running experiments with a simulated robot whose dimensions and number of IR-sensors could be identical to those of the RLAN-bot. For generalization purposes, these functions allow to change the dimensions of the robot, the number of sensors, their range and the distribution by the robot. Altogether, the following MATLAB-functions were developed:

- WheelMotionGenerator - it allows to create the different actions that the robot can choose, giving to each of them different values of angular and linear velocity.
- LineLineIntersection - it allows to find the intersection points among all lines of the environment and robot.
- IRSensorData - it allows to extract the intersection points between robot and environment and to calculate the sensor distances.
- getSensorLevels - given the distances measured by the sensors and the predefined distance levels, returns the level of each sensor in order to determine the state where the robot is.
- get_State - it allows to determine the current state of the robot taking into account the levels of each one of the sensors.
- GraphicalAnimation - it is responsible for showing the user the behavior of the robot in the environment throughout the simulation.
- MazeSimulator - it is the main function, where all the previous functions are used and where the learning algorithm is processed. It is also in this function that all the parameters are initialized.

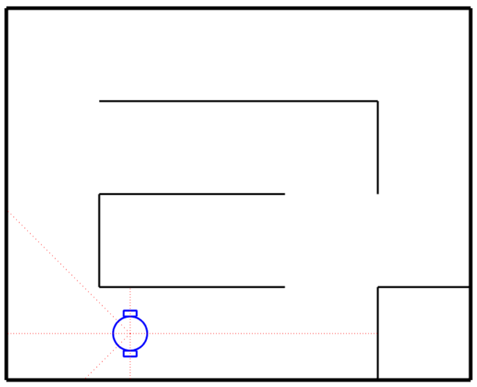


Figure 3.15: Graphical interface of the kinematic simulator in MATLAB.

Chapter 4

Proposed Methodology

This chapter presents and discusses the methodological approach adopted in this work for solving the challenges of mobile robot navigation in a maze-like environment using reinforcement learning. First, it emphasizes the problem and the principal considerations underlying the choice of HRL approach to be employed. Second, it explains how the navigation task is decomposed by introducing hierarchy into the problem. Finally, it discusses the efforts made to automatically specify the reward function.

4.1 Problem Description and Initial Considerations

This dissertation addresses the problem of a mobile robot navigating in a maze-like environment from a particular perspective that was inspired by human topological navigation and hierarchies. The following example shows how to get to the University of Aveiro (goal location) from Porto (starting location) and it helps to explain the navigation strategy to follow:

> *Take the Freixo bridge towards the Carvalhos tollgate to enter the A1, exit to "Aveiro" and, after the toll, exit on the right to enter the A25; take the A25 to "Aveiro Centro" and at the first roundabout take the second exit, follow to the traffic lights and, when there, turn right.*

This example represents a topological map that abstracts a continuous spatial experience of the driver into a discrete sequence of topological nodes, which will be called checkpoints. This map provides the connectivity among checkpoints, and thus, it identifies the underlying decision structure of the environment. These checkpoints correspond to the robots arrival at or departure from discrete areas, such as hallway intersections, roundabouts, traffic lights or visually distinct landmarks. By separating the local from the global structure of the environment, the mapping problem can be divided into smaller and easier to solve parts.

On the learning side, a topological map provides a useful representation for a hierarchical structure in which the problem can be divided into:

1. Higher-level learning of the optimal decisions (options) to be taken at the discrete checkpoints;
2. Lower-level learning of the optimal behavior (sub-tasks) between two checkpoints (optimal, for example, in terms of safety, energy consumption, etc).

As a result, the size of a topological map depends on the structure of environment itself, but not on the trajectory the robot takes nor on the time spent in the environment. As the robot travels through the environment, the model for each situation in the map can be improved as additional observations are made.

These initial considerations aimed at the careful engineering of the hierarchical framework for solving the problem at hands. In this case, the topological structure (i.e., the discrete checkpoints) consists of barcodes properly placed in the environment. The IR line tracking sensors are adapted to scan the barcodes. Fig. 4.1 shows the real maze layout (left) and the simulated one (right), where the location of the checkpoints is associated with the existence of doors in the environment.

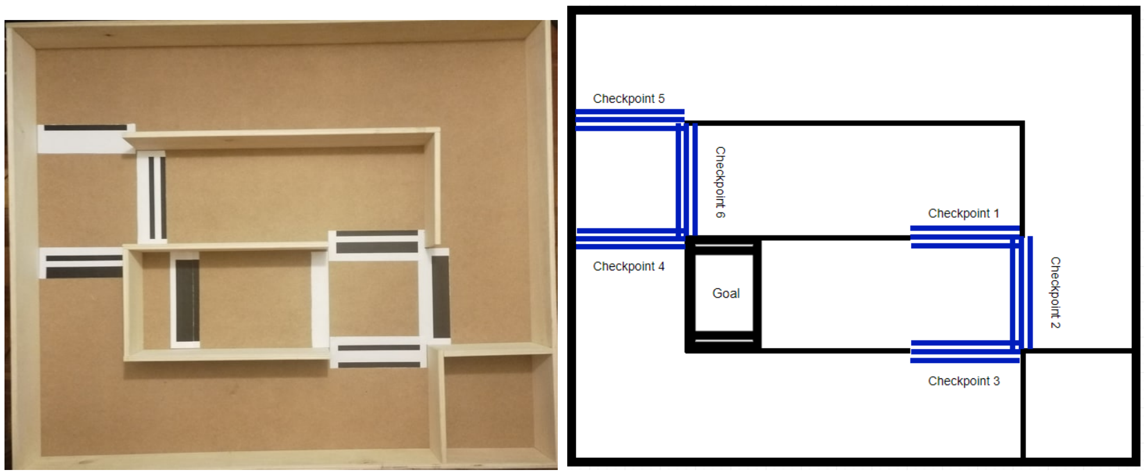


Figure 4.1: Real and simulated maze layouts with checkpoints.

The maze has a total area of (1.2 x 1.5) m2, the corridors have a width of 30 cm (15 cm height in the real 3D version) and six distinct barcodes are associated to each door. In addition to being distinguishable from one another, these barcodes should allow scanning in both directions for differentiate between arrival and departure situations. As such, their creation obeys the following rule: all checkpoint start with a black bar and end with a white bar (see Fig. 4.2). Each bar code consists of 5-bars of 1.7 cm width that occupy the width of the corridor. The goal location is associated to another checkpoint that informs the robot has reached it.

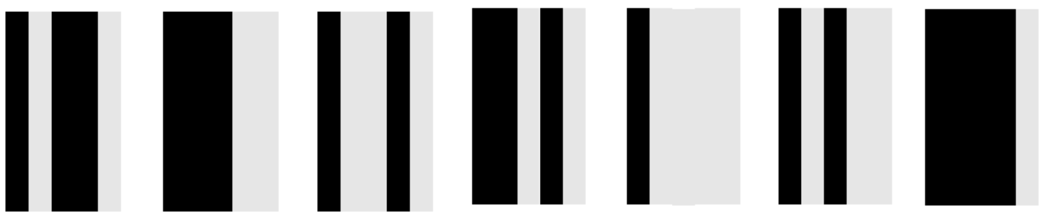


Figure 4.2: Barcodes placed on the ground environment (the barcodes associated with checkpoints 1 to 6 are represented from left to right; the GOAL checkpoint is the rightmost).

After defining the initial robots location, the robot should learn how to reach the goal location as fast as possible. Here, most of the work focuses on optimizing the performance of

RL algorithms from simulation models, before transferring the best solution on the physical system. The initial hypothesis is that, in the context of a HRL approach, robot simulations can provide a policy that, while not optimal, achieves the desired objective without a clear reality gap.

At the same time, to find a good policy it will be used a valued-based method like Q-learning (TD-learning) to measure how good an action is in a particular state. The solution adopted considers tabular forms of the state-action values, reward and policy in order to represent what is learned. Given its intuitive geometry and its relevance to maze navigation, the next section introduces the Gridworld problem proposed in [18]and solved in simulation using the Model-free algorithm, Q-Learning.

4.2 Gridworld

Gridworld is a 2D map of customizable size in the form of a maze, where the agent begins in an initial cell and aims to reach the goal cell using the minimum of possible iterations discovering the optimal policy. On the way the agent has obstacles that make the arrival to the final objective more difficult.

Hyper-Parameters

During learning, the agent wants to get the actions at a certain state that offers him the highest reward. However, for this to be possible, the agent needs to have taken all actions, so that he knows which of them is best to take in each specific situation, that is, he needs to **explore** all possible scenarios. Having all the information, the agent can choose to perform the action that gives him a higher chance of success, **exploitation**. In Reinforcement Learning, a trade-off between these two concepts, exploration and exploitation, is necessary in order to maximize the cumulative sum of rewards.

Reinforcement Learning has two distinct methods: model-free and model-based. For the problem in question, the model-free method works perfectly and so the algorithm used is the famous Q-Learning. The algorithm of Q-Learning has hyper-parameters that can not be chosen in a light mood: the learning rate, discount factor and the action selection probability.

The learning rate, or α, allows one to determine how important the new value obtained is in comparison with the previous one. The learning rate can be any value between 0 and 1 but must be chosen with discretion. If it is equal to 0, the agent has no exploratory side and does not learn with the new interactions, that is, the exploit side is favored in its entirety. If the learning rate is equal to 1, the agent forgets the previously acquired information. Thus, it can be seen that the higher the learning rate the faster the learning process, however, it may not converge correctly while using a value close to zero the convergence, although more time consuming, it is more correct. With this in mind, it is usual to give a low value to the learning rate,like 0.1, for instance.

The discount factor, or γ, shows the importance of future rewards. It is, like the learning rate, a value between 0 and 1, and if it is equal to 0, the agent is only interested in the reward obtained in the current state. For example, in the pole-balancing exercise the actions that the agent takes only need to influence the reward of that instant, so the discount factor can be equal to 0. Normally, this does not happen, so the discount factor is a value closer to 1 , often given the value of 0.9, 0.95 or 0.99, so that the reward of the next state has more weight in the update of the values.

At the moment of choosing which action to take, there must be a trade-off between exploitation and exploration, because if the chosen action is always random, the agent never acts according to the knowledge obtained, only by exploring the environment, but if the chosen action is always the one with the highest value, the agent may not be opting for the best action and is not exploring the rest of the possibilities. When choosing an action that seems to be the best, the other possible actions are not used by missing the opportunity to have better results and not completing the Q-Learning table for most values. The most commonly used methods for deciding what action to take are ϵ-greedy and Softmax. In the case of ϵ-greedy, the algorithm chooses the most valuable action in most iterations, favoring the exploitation, however, it can also explore other actions, and the probability of this happening varies depending on the value given to epsilon. The larger the epsilon, the greater the exploration and the smaller the exploitation, and obviously, if the epsilon has got a low value, the agent privileges the exploitation to the exploration. ϵ is a value between 0 and 1 as well as the learning rate and the discount factor. When choosing the action, the agent chooses the one with the highest Q-Value with a probability 1 - ϵ, otherwise it chooses a random action, which allows the optimum policy of the problem to be obtained.

The ϵ-greedy action selection has a problem: when it comes to choose an action, it is chosen completely at random and may be the worst possible action or one of the best. The solution to this disadvantage is to vary the probability of each action using a graded function. This allows to have the most valuable action with a greater probability of being chosen but also allows to rank the other actions giving a lower probability to the worst actions to take. This alternative has the name of Softmax [54], and uses the Boltzmann distribution function to assign the probability π(s$_t$, a) to the actions, Equation 4.1.

$$\pi(s_t, a) = \frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}} \tag{4.1}$$

τ is a positive parameter defined temperature. If the value is high, the actions will have equivalent probability values, otherwise the probability values are quite different. Studies done between both methods show that Softmax is better, however the parameter to be varied is very difficult to choose in order to obtain the best results. The e-greedy has the advantage of being easy to tune and allows good results aswell [55].

Knowing the effect of each of the hyper parameters of this algorithm is necessary to realize how this one works. Q-Learning is an off-policy TD control algorithm where Q is the action-value function that attempts to directly approximate the optimal action-value function, Q*, regardless of the policy followed. It is a simple algorithm but it allows a fast convergence. At each iteration, a state-action pair is visited and consequently updated by the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * \underset{a'}{max} Q(s',a') - Q(s,a)] \tag{4.2}$$

Where:

- α is the learning rate.
- r is the reward made by the action taken at the current state.
- γ is the discount factor.

- $\max_{a'} Q(s', a')$ is the maximum expected future reward given the next state and all the possible actions.

Fig. 4.3 shows several mazes that were created with a constant size (12 rows * 12 columns). The agent manages to obtain the optimal policy for all combinations of initial and final position. The agent could perform four actions (up, down, left and right), so the table Q(s,a) was a table of 144 states each with 4 possible actions. Unsurprisingly, states that create obstacles and the outer walls of the maze never have their values updated because they can not be visited by the agent.

Whenever the agent reached the goal state, the process started all over again but in an initial state randomly chosen so that all the states of the maze were updated and to prevent any state from being trapped in an action that was not the best.

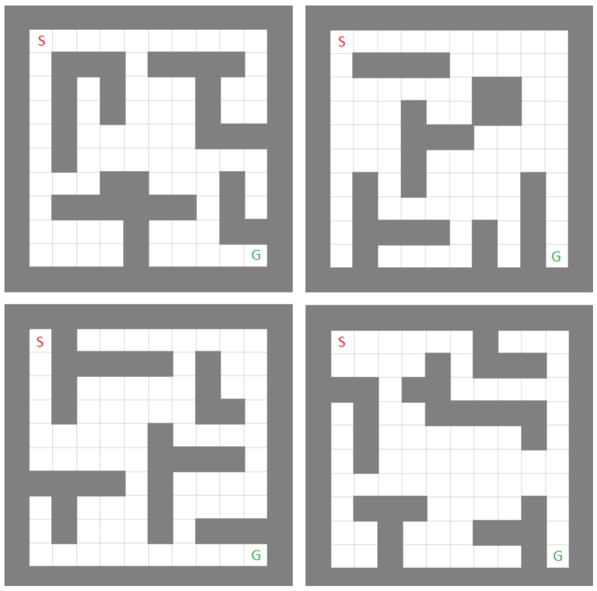


Figure 4.3: Mazes to be solved using Q-Learning.

The reward function is one of the most important aspects of the Reinforcement Learning that can be discrete or continuous. In each action-state pair the agent receives a reward that will be used to update the values of the Q-Learning table.

In this work, when the agent took an action that would cause him to hit a wall or obstacle, he had a negative reward of -100, and when he reached the goal he received a positive reward of 100. At each iteration, whatever action he took, the agent had a negative reward of -1 so that the path from the initial state to the target state is performed in the minimum possible iterations using the fastest path.

The tests were performed with several pairs of different values of learning rate (0.1, 0.2, 0.3, 0.4, 0.5) and discount factor(0.9, 0.95, 0.99), however, a research had already been carried out on which values should be used for each of the variables. The information that was collected in each performed test served to generate some graphs to understand the behavior of the algorithm used (Fig. 4.4). The graphics created show:

- The number of iterations required for the agent to arrive from the initial state to the goal state in each episode, showing a clear decrease in the number of iterations as the number of episodes increases.

Algorithm 3 Q-Learning used to solve the gridworld

```
 1: Initialize Q(s,a) with zeros
 2: Define max episodes to 500
 3: Initialize reward sum and iterations equals to zero
 4: repeat
 5:     while state != goal state do
 6:         Choose the action, a, from the state, s, using an e-greedy approach
 7:         Take the action, a, and observe the reward, r, and the next state, s'
 8:         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * \underset{a'}{max}Q(s',a') - Q(s,a)]$
 9:         Update the state
10:         iterations = iterations + 1
11:         reward sum = reward sum + reward, r
12:     end while
13: until max episodes
```

- The cumulative sum of iterations per episode, it is apparent that in an initial phase there is an abrupt growth of this, and as the agent finds an optimal policy, the number of iterations is constant, making the rise of its cumulative sum softer.
- The cumulative sum of reward per episode, where it can be seen that in an initial phase, as the agent is in a tremendously exploratory phase this sum is quite negative, and that as the number of episodes increases, the sum becomes equal to the final reward minus the number of steps taken, and the agent stops hitting walls and obstacles.

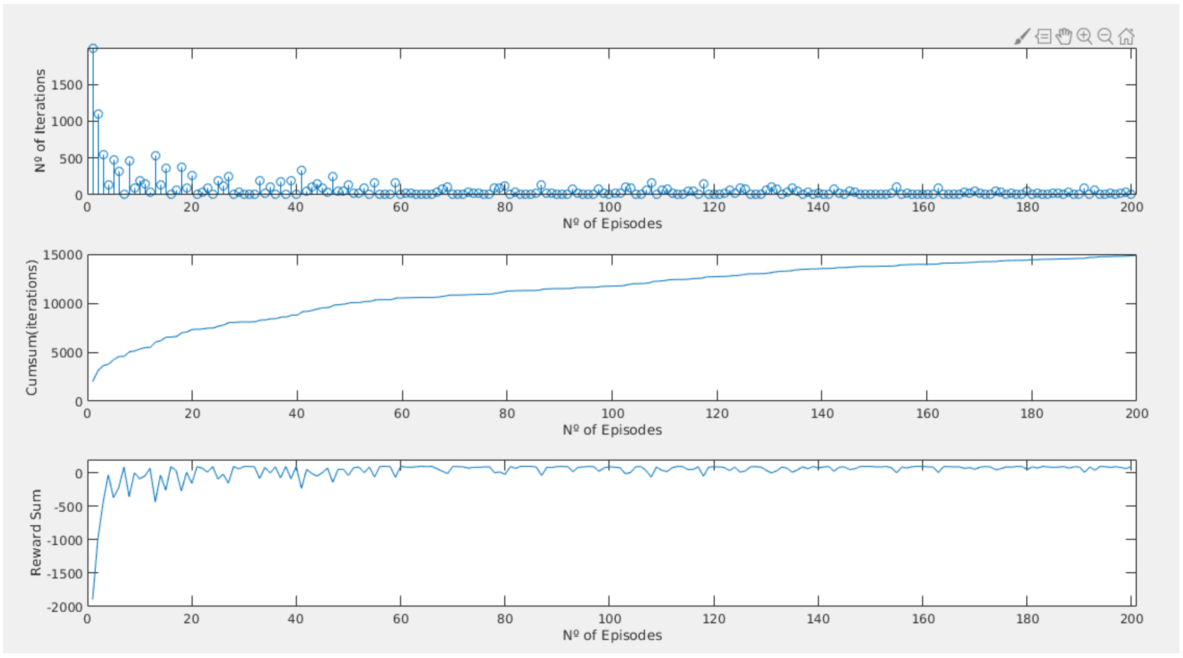


Figure 4.4: Gridworld analysis with $\alpha = 0.1$ and $\gamma = 0.95$.

Lastly, the optimal policy was obtained and several tests were carried out with random initial states to know if the agent had learned to reach the goal state in the minimum steps possible without ever being locked in an infinite loop between two states (Fig. 4.5).

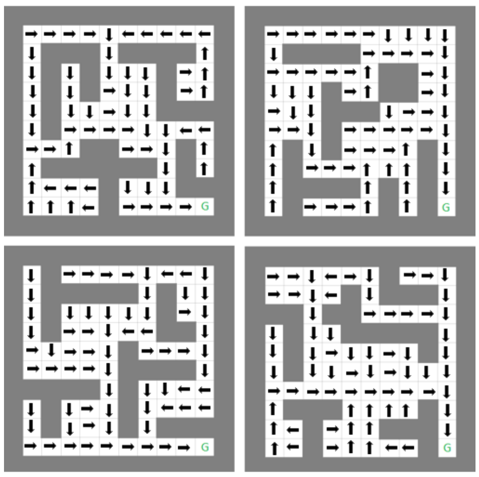


Figure 4.5: Optimal Policy for each of the mazes.

4.3 Hierarchical Approach

4.3.1 Hierarchical Decomposition

The robot system is trained using a hierarchical approach in which low-level components (sub-tasks) are sequenced at a higher-level. The proposed HRL consists of two layers (two-level hierarchy). An exercise of this kind requires an approach that divides the problem into smaller, easy-to-solve problems. The lower level is in charge of dividing the maze into distinct sub-tasks, each having its own local policy. The focal points of a maze-like environment are corridors, corners and intersections, and at a intersection there is the possibility to proceed to two or more directions (front, left or right). Thus, lower-level learning was divided into four sub-tasks. The higher-level is in charge of making the decision, that is, choose an option for the sub-task that will perform at the entrance of a checkpoint, and the only sub-tasks that are considered in this moment of decision are the sub-tasks inherent to the passage through a door. However, when one of the checkpoints is read in reverse, the robot knows that it is in a corridor or corner, using the local policy of this sub-task (see Fig.4.6).

The algorithm used at the higher level is the classic Q-Learning in which actions are considered as options, which represent the set of actions to be performed on arrival at a door, whether it is moving forward, right or left.

Using the idea provided by the Options framework in which an option is a set of individual actions resulting in an abstraction of states where the lower level is a sub-policy where the output is an action and the goal is to reach the end of each sub-tasks and the higher level chooses which of the sub-policies to use at each moment that an option is needed. The start and end of each sub-task is done using checkpoints, a feature that is not used by any of the previously mentioned algorithms. Taking into account the HAM algorithm, some ideas were also taken from its concept. Whenever the agent reaches a door it is in the Choice state, where it can choose one of 3 options. The purpose is to know which option, in this concrete

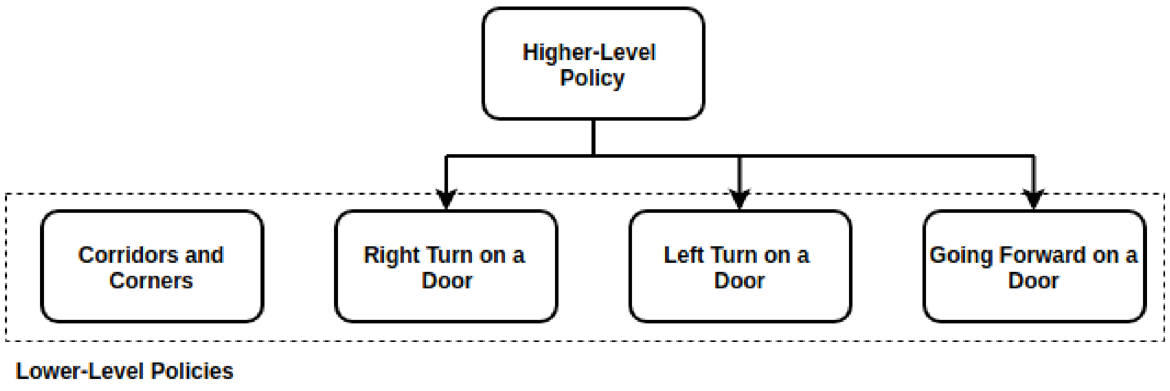


Figure 4.6: Two level hierarchy flowchart.

case called the machine, to call and with what probability. The optimum policy is obtained recursively, as in the case of the MAXQ algorithm as opposed to the Options Framework and HAM where it is obtained hierarchically. In the case of the MAXQ algorithm there is an optimal policy for each of the sub-tasks. The algorithm used allows a temporal abstraction and a convergence for optimal solution such as all the frameworks presented in chapter 2 and allows to obtain an optimal policy for each of the sub-tasks as well as an optimum policy for the higher level of the problem.

4.3.2 Learning of Lower-Level Sub-tasks

Each of the sub-tasks has a state space and an action space required to complete their learning tables. Before beginning the learning, it is necessary to initialize some parameters that are known a priori: number of states and numbers of actions that serve to define the size of the table of Q-Learning. Since the built robot has six infrared sensors for distance measurement, four levels have been defined to agglomerate the measured values at each moment of each of the sensors, Table 4.1. Having four levels of distances for each of the six sensors, it is possible to obtain the total number of states: 4^6 = 4096 states.

Measured Distance (cm)	**Level**
$0 \leq$ d < 7	0
$7 \leq$ d < 13	1
$13 \leq$ d < 20	2
d ≥ 20	3

Table 4.1: Discretization of measured distances.

The discretization of the distance measured by each sensor is made to reduce the complexity of the problem and the total number of states. If this was not done, each infrared sensor, which has the ability to measure distances between 4 and 30 cm reliably, would have 26 possible levels. With 26 levels per sensor, the total number of states would increase to 26^6 = 308915776. This number is excessive for the experiment to be carried out so the discretization on four levels seems to be much more plausible. The robot does not have access to its global position in the maze, since the information obtained by the infrared sensors only offers its local position, showing the distance to the walls of the maze. This property makes this experiment different from the Gridworld case studied earlier, where the agent had access to their global position and where the states were that same position.

After knowing the total number of states, it is necessary to define the number of actions achievable by the agent knowing that in a maze the robot needs to change direction whenever approaching a corner or whenever it arrives at a door and needs to turn either to the right or to the left, and also needs to compensate for a possible approach to a wall. With this in mind, the robot has 9 possible actions where one of them is a motion with an angular velocity equal to zero and constant linear velocity, and the remaining eight have different angular velocities that allow the robot to turn both left and right with more or less smoothness (Fig. 4.7).

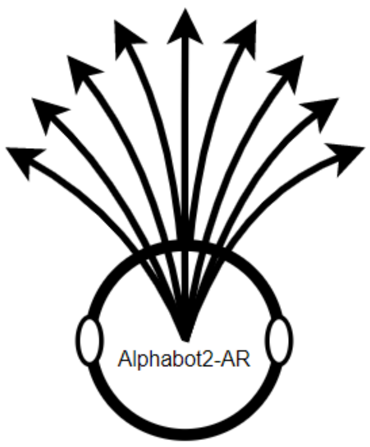


Figure 4.7: Possible actions of the robot.

Having acquired the information of the number of states and number of possible actions in each of the sub-tasks, it is possible to initialize the Q-Learning table for each one, Table 4.2. At the beginning of the learning these tables are initialized to zero and as the agent is in a state-action pair, the value corresponding to that action-state pair in the table is updated according to the equation 4.2.

State\Action	a_1	a_2	... a_j ...	a_9
s_1	$Q(s_1,a_1)$	$Q(s_1,a_2)$	... $Q(s_1,a_1)$ ...	$Q(s_1,a_9)$
s_2	$Q(s_2,a_1)$	$Q(s_2,a_2)$	... $Q(s_2,a_j)$ ...	$Q(s_2,a_9)$
.	.	.	. . .	.
.	.	.	. . .	.
s_i	$Q(s_i,a_1)$	$Q(s_i,a_2)$	.. $Q(s_i,a_j)$ ..	$Q(s_i,a_9)$
.	.	.	. . .	.
.	.	.	. . .	.
s_{4096}	$Q(s_{4096},a_1)$	$Q(s_{4096},a_2)$	... $Q(s_{4096},a_j)$ ...	$Q(s_{4096},a_9)$

Table 4.2: Q-Learning table for each of the sub-tasks.

The reward function is different for each case, because in a sub-task there are states that should privilege a certain action that is not the same in one of the other sub-tasks. For this, the reward function was created in an intensive way so as to make a dedicated learning for each sub-task. This intensive study was performed in simulation seeing the particularity of each state and determining a positive reward equal to 1 for the most beneficial action for the robot, and negative rewards for all the other eight actions.The most negative rewards were used for the actions that would take the robot to a more distant position from the one that is desired. This process allowed a much faster learning and a very effective and smooth drive through the maze. Seeing as an example the state 4064, presented in Fig. 4.8, in the sub-task "Right turn on a door" the action to be privileged is the action with the highest value of angular velocity to the right (action 5), but if the sub-task is the "Left turn on a door", this state already needs to focus the second smoother action to the left (action 7). The reward function of this state in the sub-task "Right turn on a door" is [-3 -3 -3 -3 1 -5 -7 -9 -10], and in the sub-task "Left turn on a door" it is [-3 -5 -7 -9 -10 -3 1 -3 -3]. The particularity of this type of reward is that it is modeled from the previous knowledge that the user has about each state. Although it is a valid solution, there are other more automatic alternatives that will be discussed later in this chapter.

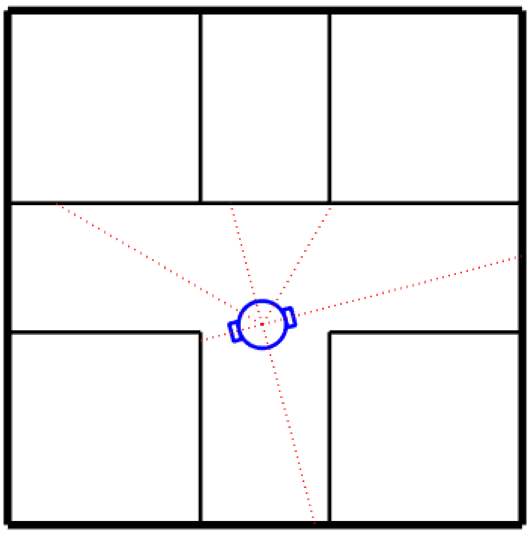


Figure 4.8: Representation of the state 4064.

For a more complete learning of each sub-task, several tests were performed on each one, beginning with different positions and orientations on the environment.

In the sub-task "Corridors and corners", three simulation maps were created so that most of the possible states in this case were visited (Fig. 4.9). The last two maps were created because, during the simulation in the complete maze, it was observed a bad behavior of the movement of the robot to the exit of a corner with a door in the end, that have states that had not been contemplated in the first map.

During the process of learning the sub-tasks "Right turn on a door" (Fig. 4.10) "Left turn on a door" (Fig. 4.11) and "Going forward on a door" (Fig. 4.12) there were also three maps created for each of them to be able to contemplate all possibilities in a maze.

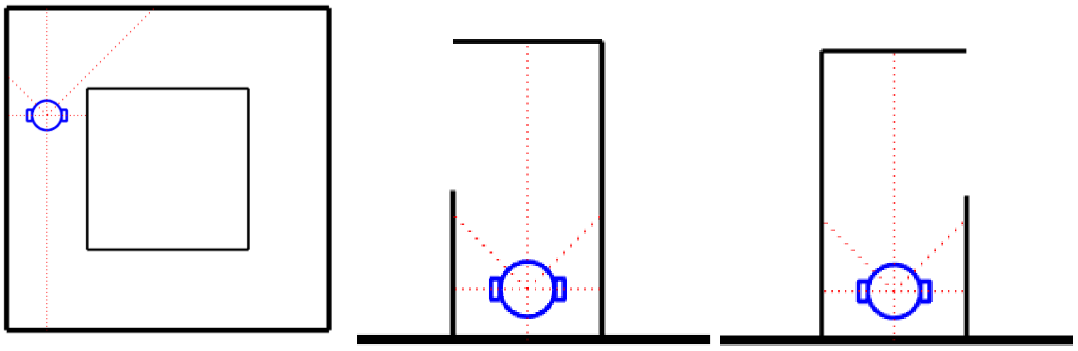


Figure 4.9: Maps 1, 2 and 3 created to simulate the sub-task ”Corridors and Corners”.

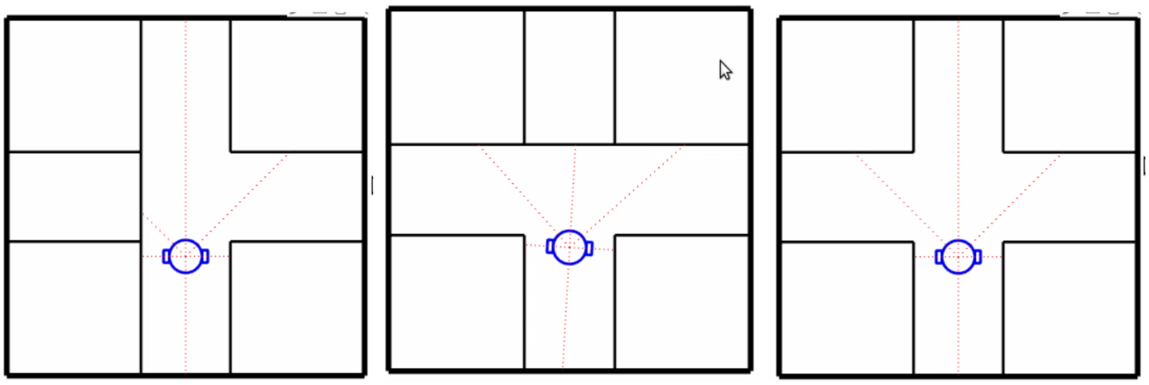


Figure 4.10: Maps 1, 2 and 3 created to simulate the sub-task “Right turn on a door”.

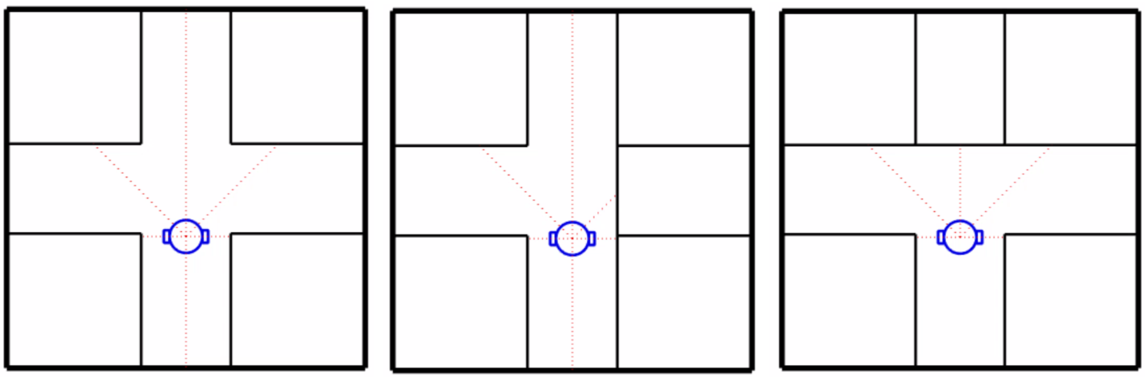


Figure 4.11: Maps 1, 2 and 3 created to simulate the sub-task “Left turn on a door”.

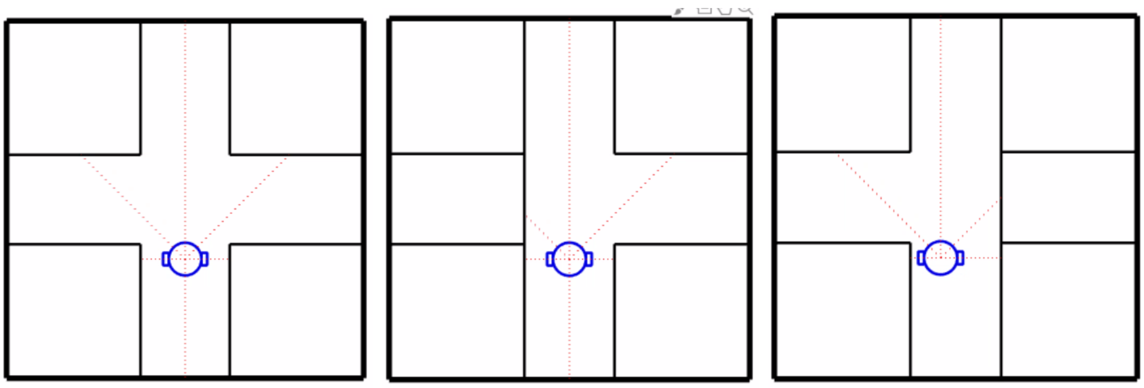


Figure 4.12: Maps 1, 2 and 3 created to simulate the sub-task “Going forward on a door”.

4.3.3 Learning of Higher-Level Options

The high level of learning allows the robot to choose the sub-task it needs to perform at a given time. For the learning algorithm, it is necessary to know the number of states and the number of options. The number of states will be the total number of checkpoints, counting with the goal state, and the number of options will be the number of possible moves on arrival at a door: turn right, turn left and go forward. Thus, there are 7 states and 3 options (see Table 4.3).

State\Option	o_1	o_2	o_3
s_1	$Q(s_1,o_1)$	$Q(s_1,o_2)$	$Q(s_1,o_3)$
s_2	$Q(s_2,o_1)$	$Q(s_2,o_2)$	$Q(s_2,o_3)$
.	.	.	.
.	.	.	.
s_i	$Q(s_i,o_1)$	$Q(s_i,o_2)$	$Q(s_i,o_3)$
.	.	.	.
.	.	.	.
s_7	$Q(s_7,o_1)$	$Q(s_7,o_2)$	$Q(s_7,o_3)$

Table 4.3: Q-Learning table for the higher level of the learning process.

Firstly, a simulation was made where the robot would navigate through the complete maze trying to find the goal, using in each of the sub-tasks the updated table of Q-Learning. This simulation was divided in 50 episodes where the start position was a random position in the maze, and the objective was to reach the goal at the end of each of the episodes. To do this task, the reward function had a simple metric: if the chosen option causes the robot to hit a wall, the reward is -10, if it does not the reward is -1, and if the robot arrives to the goal, then the reward is 10.

The second approach was used to simulate the complete maze using the updated tables for each sub-task and perform the learning process of higher level throughout the simulation. As the number of checkpoints and options is very low, this learning is very fast to converge on a solution that takes the robot to the goal. This simulation was divided in 50 episodes that finished with the arrival of the robot to the goal. The beginning of each episode was done in a random position in one of the corridors of the maze. In terms of the reward function, each option taken by the robot that does not lead him to hit a wall has a reward of $r = -1 - (numberOfIterations * 0.001)$, the opposite has a reward of -10, and reaching the goal has a reward of 10. The choice of an option was made using the ϵ-greedy method.

This process gives very good results, however, the learning algorithm for this phase can be done using a state machine as a reference as can be seen in Fig. 4.13. This representation is used to allow the algorithm to know what the next state is, as well as the reward to provide in each case of option-state pair taken at each instant and to accelerate the process of convergence to the optimal policy, however the agent receives help from the user. The negative reward equal to -1 is given if the option chosen is the one that takes the robot through the fastest route. If the reward is equal to -2, the option chosen is possible but the route made by the robot is longer. When the reward is -10, the option chosen would take the robot against a wall and could be stuck. If the robot receives a reward of 10, it has arrived to the goal, finishing the episode.

The algorithm used was the model-free algorithm Q-Learning again. Since it is an exercise with few states and few options, it does not take many episodes, as the algorithm converges quickly to an optimal solution.

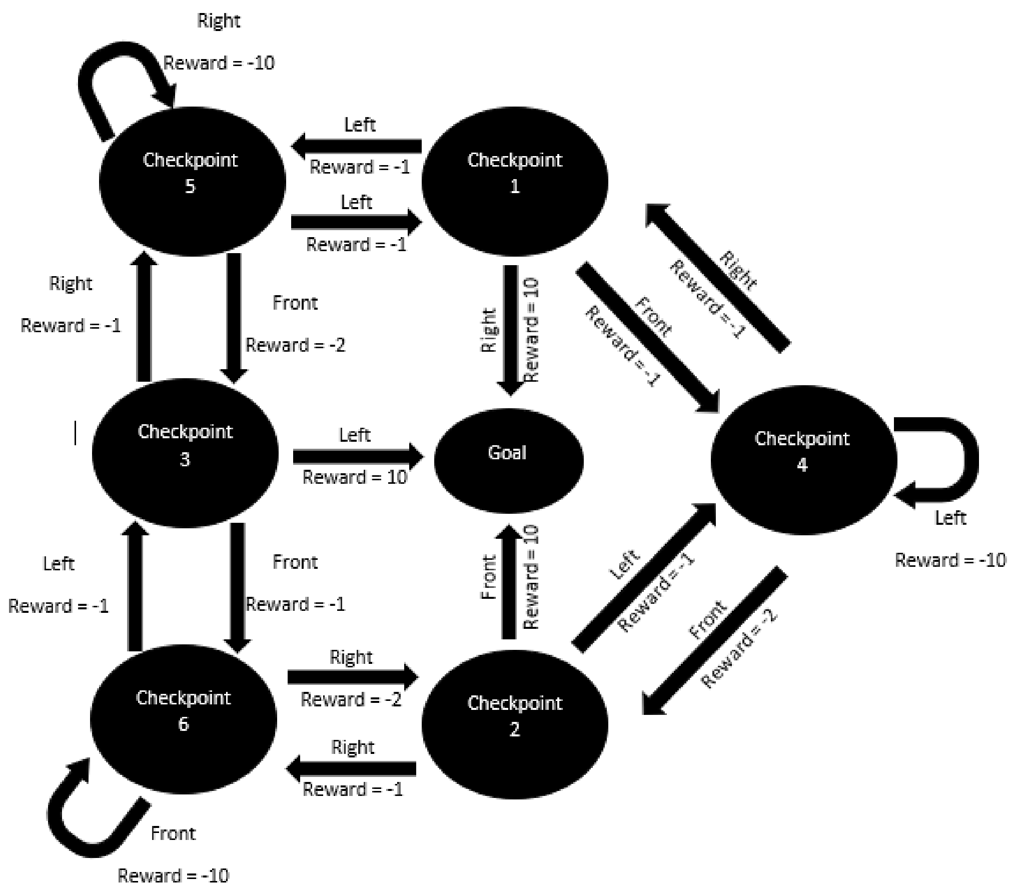


Figure 4.13: State machine used in the learning process of the higher level.

4.4 Reward Function Design

A reinforcement learning agent learns, by trial-and-error, how good or bad are its actions based on the rewards it receives from the environment. Thus, the specification of the robots behavior and goal, through a reward function, must be carefully designed in order to achieve the ultimate goal of maximizing the accumulated long-term reward. Although in some domains it seems natural to provide rewards upon task achievement, the most frequent is to include intermediate rewards guiding the learning process to a desired solution.

Reward shaping is a process by which the system gathers the notion of proximity to the desired behavior, instead of simply encoding success or failure [56]. In this context, Inverse Reinforcement Learning (IRL) is the problem of learning a reward function assuming that this can be reconstructed from expert demonstrations [57, 58]. The design of the reward function in terms of some selected features, which depend on the robots action and the state of the environment, can be an effective procedure [59].

Here, two alternative approaches for specifying the reward function will be compared. First, a manual specification of the reward function is conducted, requiring exhaustive trial-and-error for properly tuning the values making the robot follow aligned with the corridors center line, turn in a corner and cross a door. This specification of the reward function is time-intensive and cumbersome, being included only for comparative purposes. The difficulty also results from the fact that the state-action spaces in which the learning algorithm operates does not allow a straightforward representation of the desired behavior.

The alternative is an automatic procedure, running off-line, based on local features representing the robot navigation in a corridor and corner (doors are not considered here). For example, in a corridor, the local posture of the mobile robot is completely characterized by its distance from the corridor center line and the orientation as related to the corridor axis (Fig. 4.14). For each discrete state, these two features are used for mapping the particular state to a cost function that respects time and safety operation. The following subsections describe the automatic design of the rewards using the simulated robot.

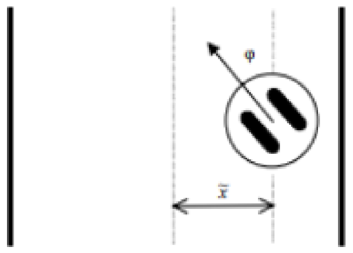


Figure 4.14: Local representation of the robot state defined by the distance to the corridors center line and the angle as related to the corridors axis.

4.4.1 Corridor

In order to obtain a reward function that is built automatically, one began by thinking how it would be possible to do it in a corridor without paying attention to the corners. For this it is necessary to make the robot know its local position in the simulated map, because in the real environment the robot only has information about its local position. Therefore, it is necessary to obtain the global and local origin coordinates of this exercise. Knowing the coordinates is necessary to make a transformation where they can be compared in the corridor (Fig. 4.15).

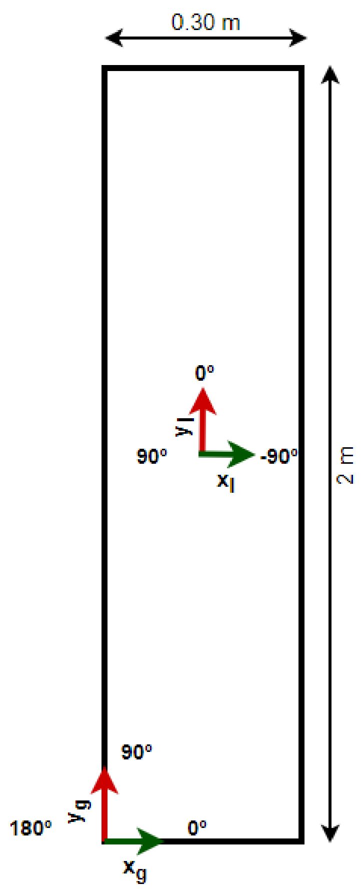


Figure 4.15: Transformation made between global and local coordinates - Corridor.

$$\begin{cases} x_g & = x_l + 0.15 \\ \theta_g & = \theta_l + \frac{\pi}{2} \\ y_g & = 1 \end{cases} \tag{4.3}$$

For that, in simulation, a closed corridor was constructed and it was concluded that there would be at least three possibilities to create the reward function:

1. Reward function dependent only on the local position x_l (regardless of the action used) (Fig. 4.16).

$$reward = \begin{cases} (-500 * x_l^2) + 0.5 & if x_l >= -0.09 \vee x_l <= 0.09 \\ -10 & else \end{cases} \tag{4.4}$$

2. Reward function dependent on the local position x_l and the orientation θ_l regardless of the action used (Fig. 4.17).

$$reward = -500 * (x_l^2 + 0.0159 * \theta_l^2) + 2 \tag{4.5}$$

3. Reward function where the result is the sum of the reward function 2 with the inclusion of an action reward function depending on where the robot is in the corridor. This reward function dependent on the action is similar to the one made before, where the user gives some help choosing the best actions to take in a particular situation.

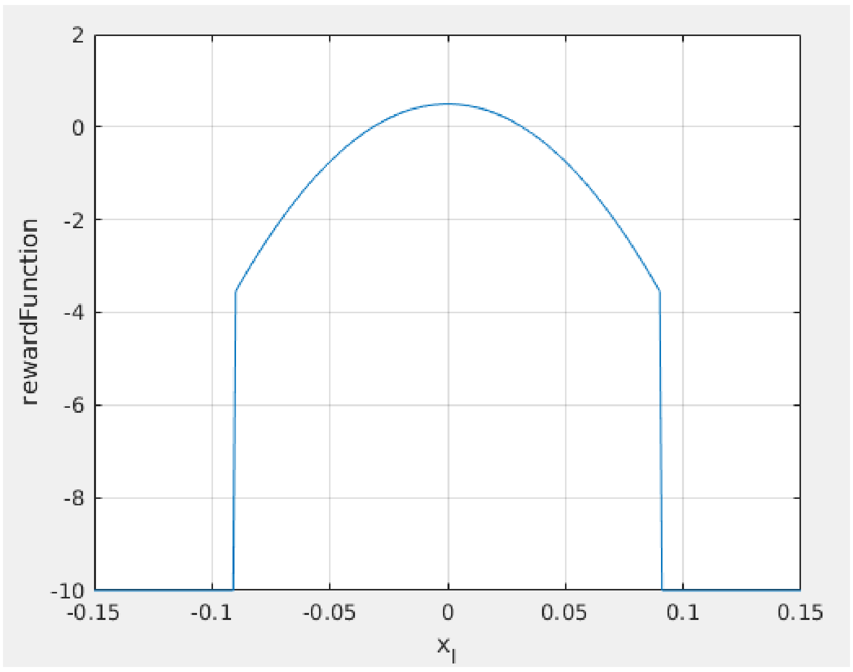


Figure 4.16: Reward Function 1.

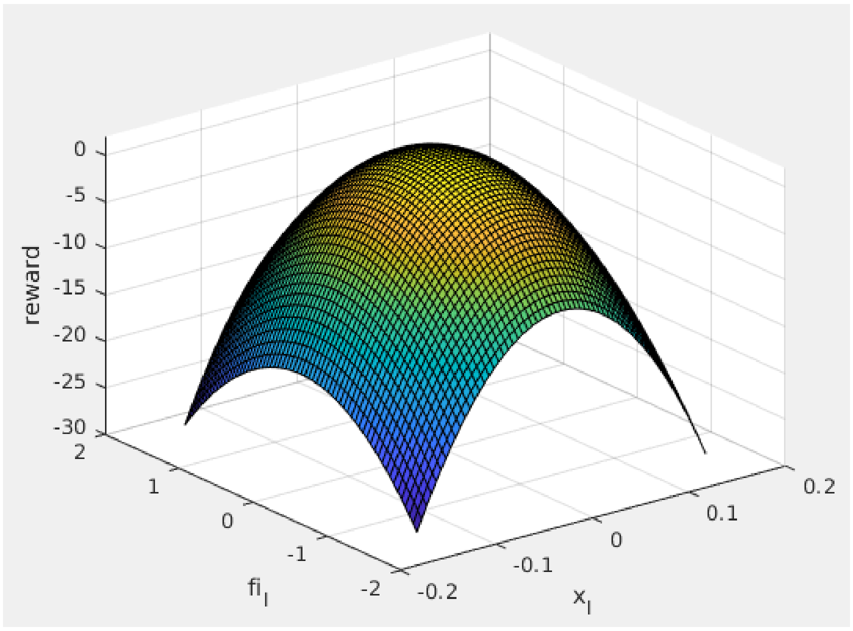


Figure 4.17: Reward Function 2.

When performing the learning process with the 9 actions (Fig. 4.7), and using the reward function 1 and 2, there are situations where the robot can not find a solution to get out of the state in which it is, as can be seen in Fig. 4.18. With this information it would be necessary to find a way to overcome this problem and the solution found was to add two more actions to the existing ones: turn itself to the right and to the left. With the inclusion of these two actions, it was possible to obtain results that were much better than those previously obtained, and that the robot finally managed to learn cases that were previously impossible to solve. However, the first two reward functions still had problems in certain cases and did not converge to an optimal policy.

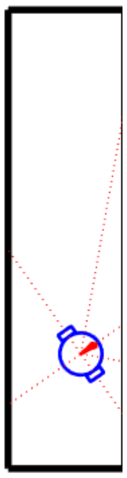


Figure 4.18: Proof of insufficient number of actions.

4.4.2 Corner

The corner is way harder to compute because there are three variables at stake: x, y and θ. In the Fig. 4.19, it is represented the transformation between the global and local positions of the robot as well as the different sections. Each of these sections has a reward function.

$$\begin{cases} x_g &= x_l + 0.15 \\ \theta_g &= \theta_l + \frac{\pi}{2} \\ y_g &= y_l + 1.7 \end{cases} \tag{4.6}$$

Each section represented in the Fig. 4.19 has a reward function to allow the robot to plan is motion the best way possible in order to drive through the corner in a smooth way:

- Section 1 - This section still belongs to the corridor so reward function is equal to the reward function 3 (Fig. 4.17) depending on the local position x_l, orientation θ_l and the action chosen.

- Section 2 - The robot has no interest in being in this corner zone so the reward function is always negative and varies depending on its local position, x_l and y_l (Fig. 4.20).

$$reward = -500 * (x_l^2 + y_l^2) - 10 \tag{4.7}$$

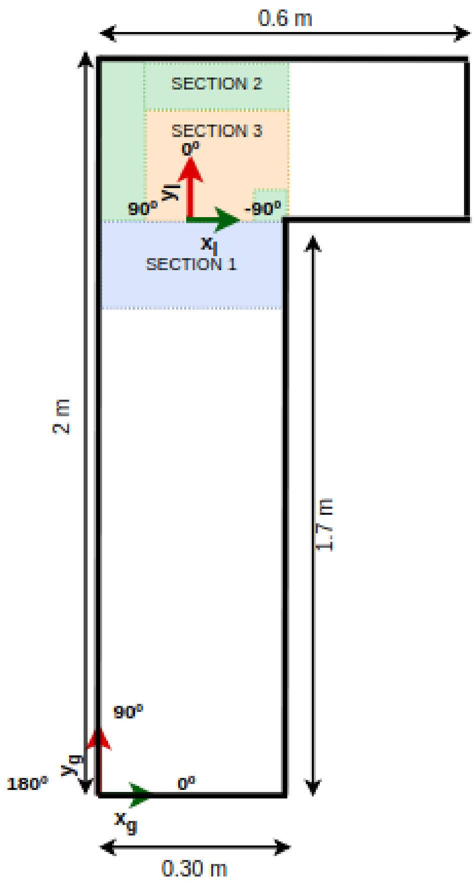


Figure 4.19: Transformation made between global and local coordinates - Corner.

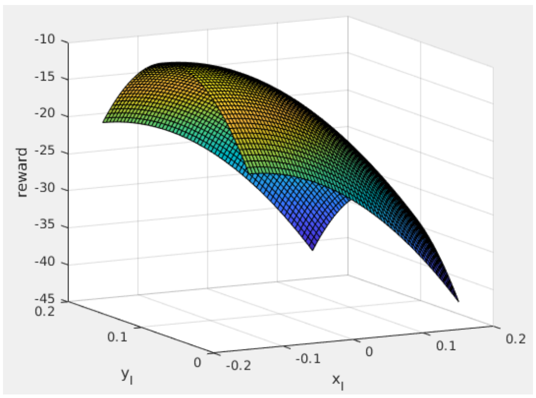


Figure 4.20: Reward Function - Section 2.

- Section 3 - It is the section of interest to the trajectory of the robot, then the reward function must be positive (Fig. 4.21).

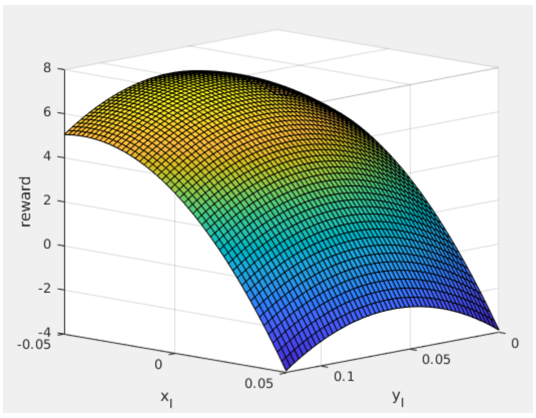


Figure 4.21: Reward Function - Section 3.

$$reward = -750 * (x_l^2 + (y_l - 0.06)^2) + 7 \tag{4.8}$$

Chapter 5

Experiments and Results

This chapter describes the set of experiments conducted in order to train the mobile robot to navigate in the maze environment. Table 5.1 discriminates the experiments performed in simulation from those carried out with the real robot, as well as the lower-level from the higher-level. The discussion of the main results is presented at the end of each subsection.

		Proposed Experiments	Section
Simulation	Lower Level	Corridors and Corners with manual reward	5.1.1
		Corridors and Corners with automatic reward	5.1.2
		Right turn on a Door	5.2.1
		Left turn on a Door	5.2.2
		Going forward on a Door	5.2.3
	Higher Level	Learn the route to the goal	5.3
		Learn the fastest route to the goal	
		State machine to learn the fastest route to the goal	
Real Environment		Learning from scratch	5.4
		Transfer Learning	
		Transfer Learning and continuous learning	

Table 5.1: Proposed Experiments

5.1 Simulations in "Corridors and Corners"

5.1.1 Manual Reward Function

The way a random action was chosen was a deliberated factor and to be able to compare what would be the most correct method that would allow to update a greater number of states, it was decided to use two different approaches. In the first one, the e-greedy method was used with a constant probability value of $\epsilon = 0.1$, and in the second one, the annealing e-greedy algorithm with $\epsilon = 1/\log(\text{iteration})$ was used, where in the beginning epsilon value is near infinite, focusing the exploration, and as the number of iterations increases, the epsilon reaches values near zero, increasing the exploitation. This also allows us to know the number of iterations necessary for an acceptable learning in each case, since it may be possible, at first, to use a large number of iterations without necessity.

In order to visualize this particularity, one can use the Fig. 5.1 where it is noticed that the number of states visited by the robot in the case of annealing e-greedy (112) is higher than the case of e-greedy (103). In this first experiment, every 50000 iterations a new position was chosen to begin with a different orientation. Consequently, the huge increase in the number of states visited when the simulation reaches 100000 iterations since previously, the robot was moving in the corridor in a clockwise direction and from that moment on, it started to move in the counterclockwise direction.

In the Fig. 5.2, it is possible to observe which the 10 most visited states during this learning are. After the learning process it is possible to see the trajectory that the robot makes along the map with access to the updated Q-Learning table (Fig. 5.3) and to demonstrate if the behavior of the robot is good, one can analyze the error of the position and orientation of the robot in the beginning, middle and after the learning process is completed in the corridors and in the corners, as shown in Figs. 5.4 and 5.5.

It is possible to see that the robot has learned to perform a regular and smooth trajectory through the corridors and corners, thus improving the behavior demonstrated in an initial phase of learning obtaining a position and orientation error fairly low.

However, it is necessary to compare the results obtained with the use of the manual reward function and the automatic reward function that will be addressed in the following subsection.

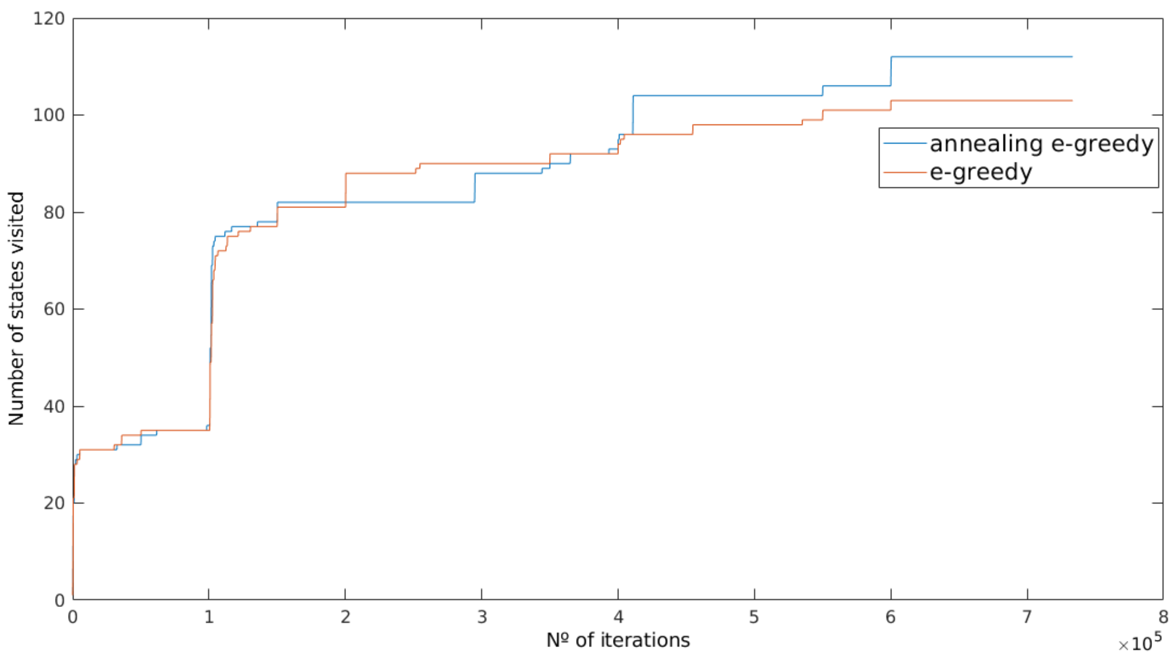
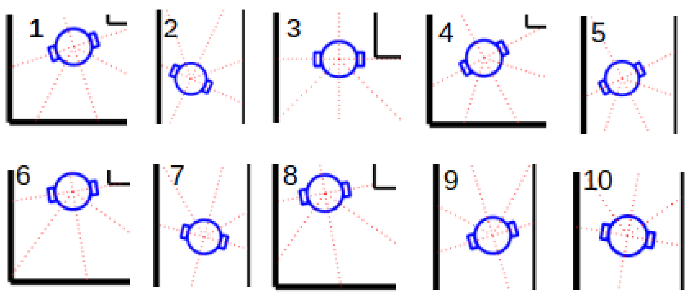


Figure 5.1: Sub-task "Corridor and corners" - states visited in function of the number of iterations.



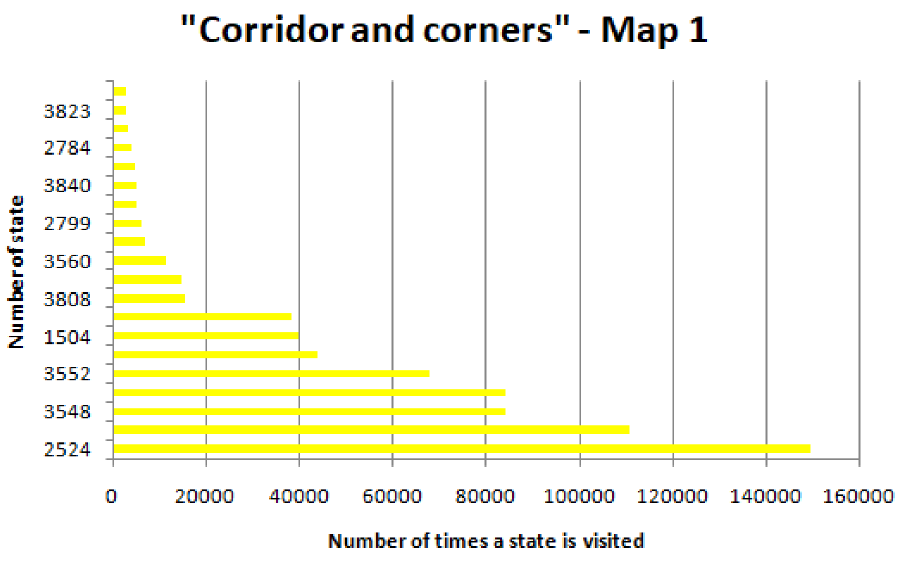


Figure 5.2: Number of times a state is visited - ”Corridor and corners”.

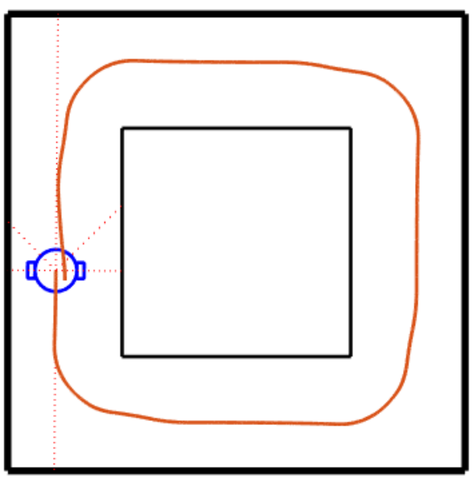


Figure 5.3: Trajectory of the robot using the updated Q-learning table - ”Corridor and corners”.

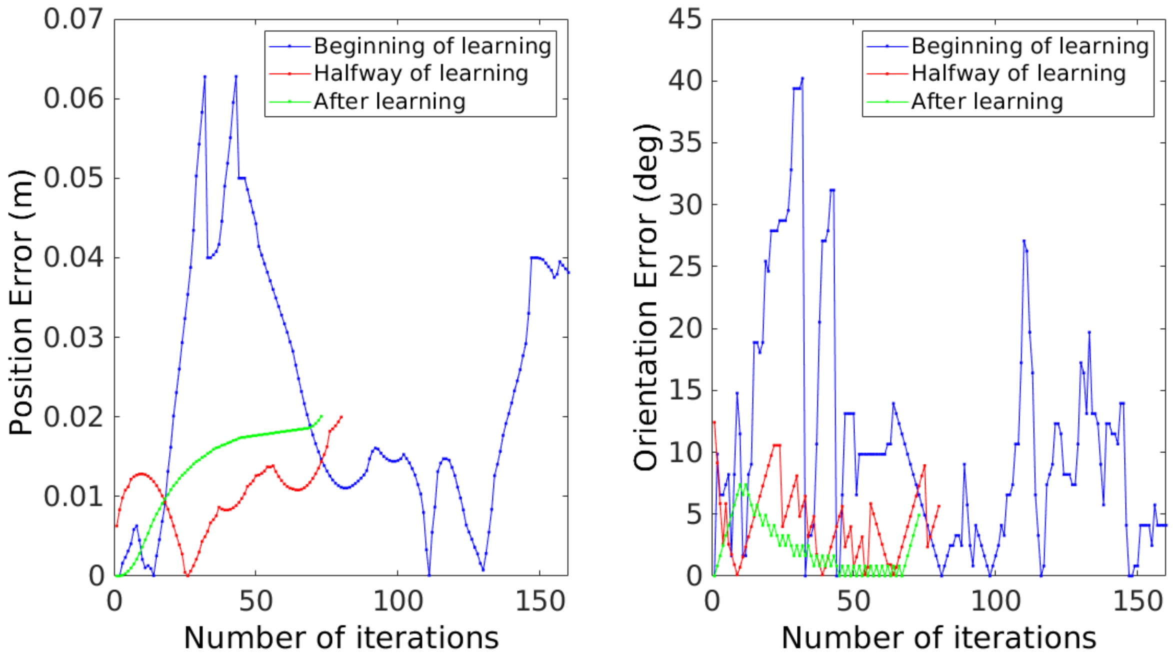


Figure 5.4: Error of the position and orientation of the robot with the manual reward - Corridor.

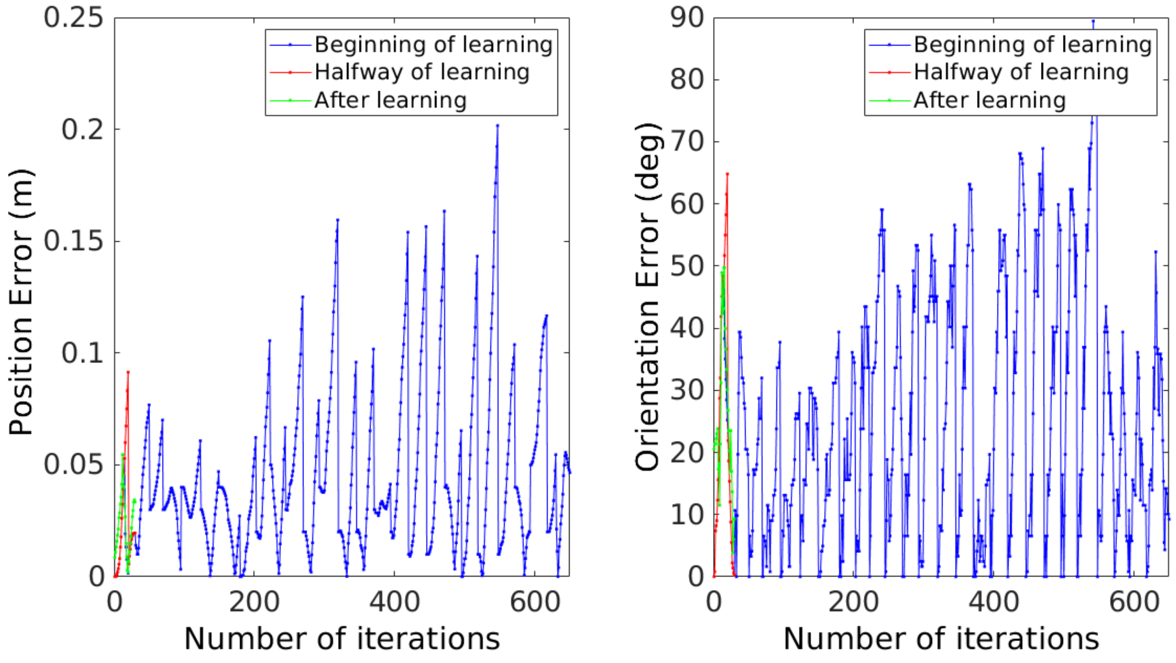


Figure 5.5: Error of the position and orientation of the robot with the manual reward - Corner.

5.1.2 Automatic Reward Function

Corridor

During the experimental phase, tests were performed with the three automatic reward functions described in the previous chapter with regard to the corridor. The third reward function proved to be the best one to solve the corridor even when the robot was facing the wall of the corridor. In the Fig. 5.6 it is possible to see that the reward function 3 allows the robot to correct its behavior even in the worst initial situations.

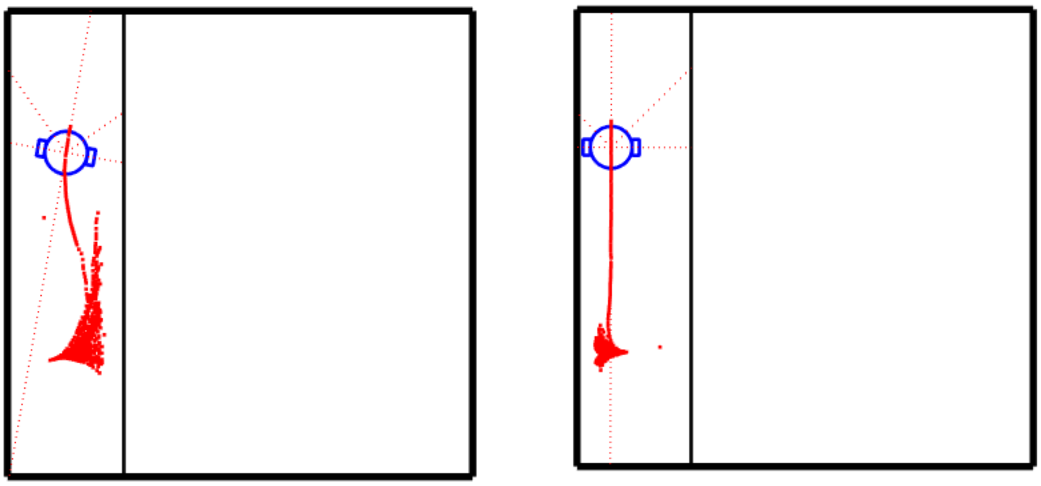


Figure 5.6: Reward Function 3 - First and second episodes of the learning process.

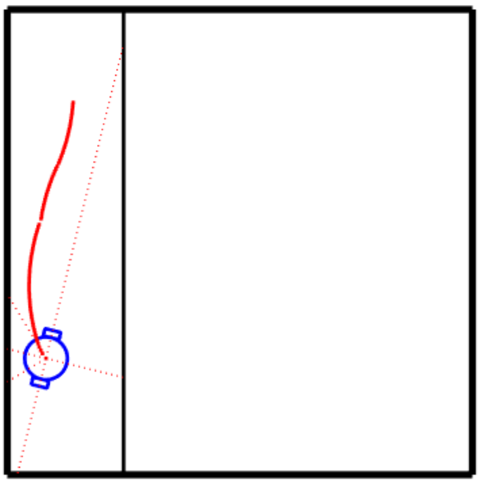


Figure 5.7: Trajectory made by the robot after the learning process using the reward function 3.

Fig. 5.7 shows that the robot can find the path to navigate through a corridor, even facing the wall initially after the learning process is completed.

The ten most visited states during the learning process can be seen in Fig. 5.8. To demonstrate if the behavior of the robot is good, one can analyze the error between the position of the robot and the center of the corridor and the error between the orientation of the robot and the desired orientation ($\theta_l = 0$) in the first and tenth episodes of the learning process and after the learning is completed as can be seen in Fig. 5.9. It is fair to say that the robot starts to learn to go near the center of the corridor and almost parallel to the walls. Comparing the position and orientation error graphs obtained with the automatic and manual reward function with respect to the corridor represented respectively in Fig. 5.9 and Fig. 5.4, it is possible to conclude that the automatic function allows the robot to have a trajectory closer to the center after learning.

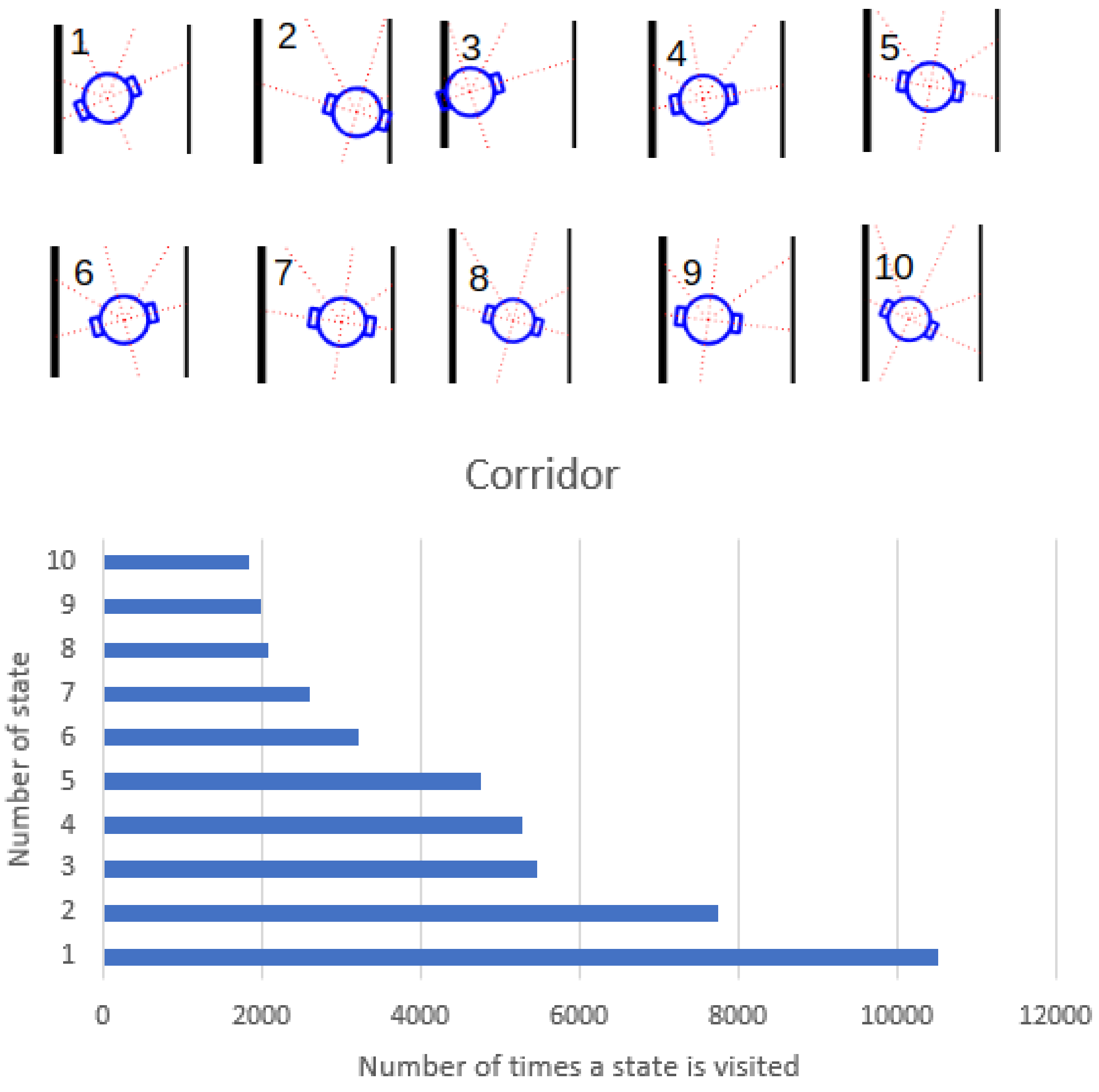


Figure 5.8: Number of times a state is visited - Corridor.

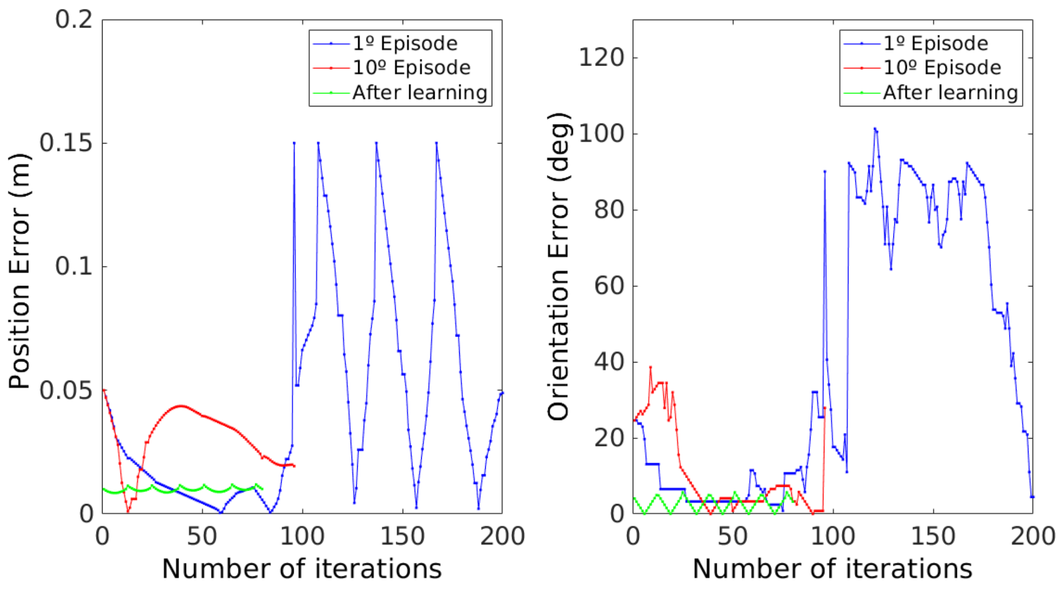


Figure 5.9: Error of the position and orientation of the robot with the automatic reward - Corridor.

Corner

After an iterative analysis of all the different positions and orientations present in the three sections, all the states present in a corner have their corresponding reward and it is possible to test the learning algorithm in the same way that was performed during the tests in the corridor. This time, the eleven actions were used as in the last test done in the corridor, and the results can be observed in the following figures 5.10, 5.11 and 5.13. The ten most visited states during the learning process can be seen in Fig. 5.12.

Analyzing the graphs of the error of the position and orientation in the case of a corner using the automatic reward function, as shown in Fig. 5.14, it is possible to conclude that at the beginning the process is very time consuming until reaching the goal, however, as the episodes pass the robot becomes able to complete its task quickly and smoothly. Compared to the graphs of the error of the position and orientation using the manual reward function, as shown in Fig. 5.5, the robot does not need as many iterations at the beginning of learning to reach its goal.

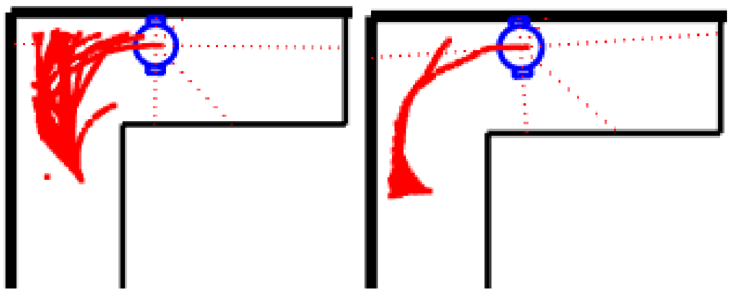


Figure 5.10: Corner - First and second episodes of the learning process (Direction 1).

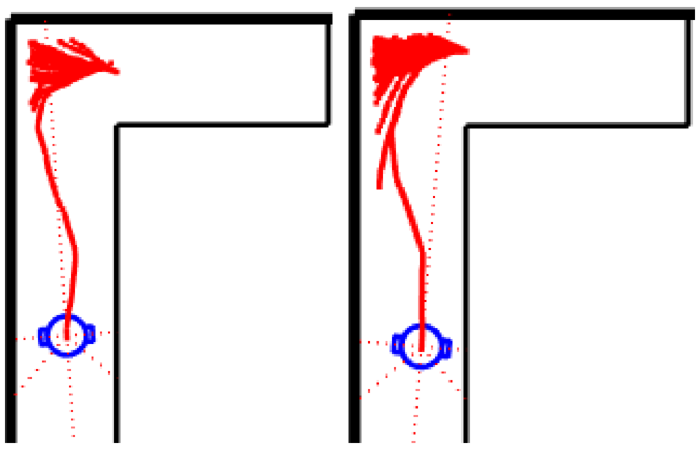
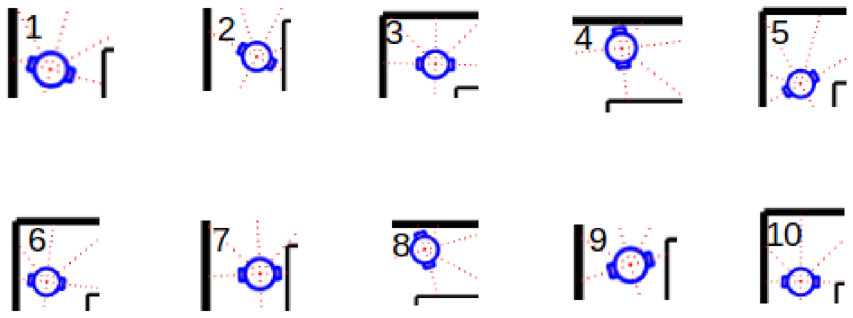


Figure 5.11: Corner - First and second episodes of the learning process (Direction 2).



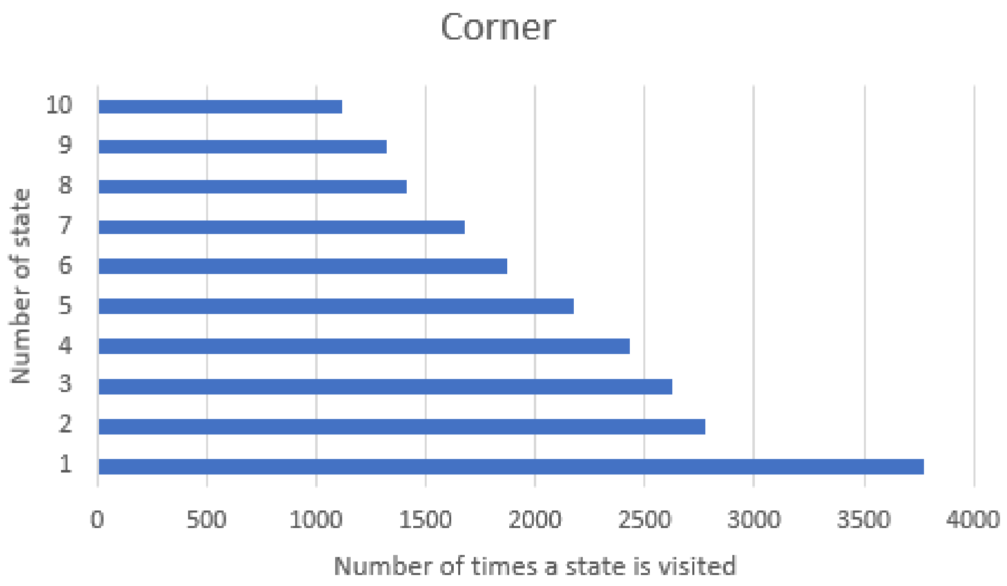


Figure 5.12: Number of times a state is visited - Corner.

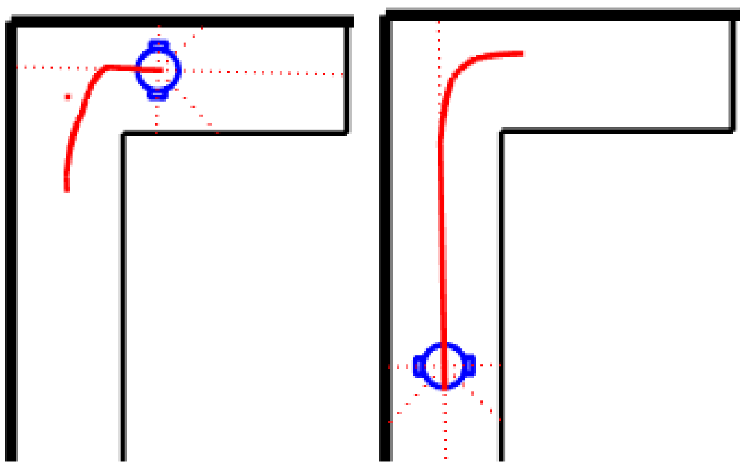


Figure 5.13: Corner - After the learning process.

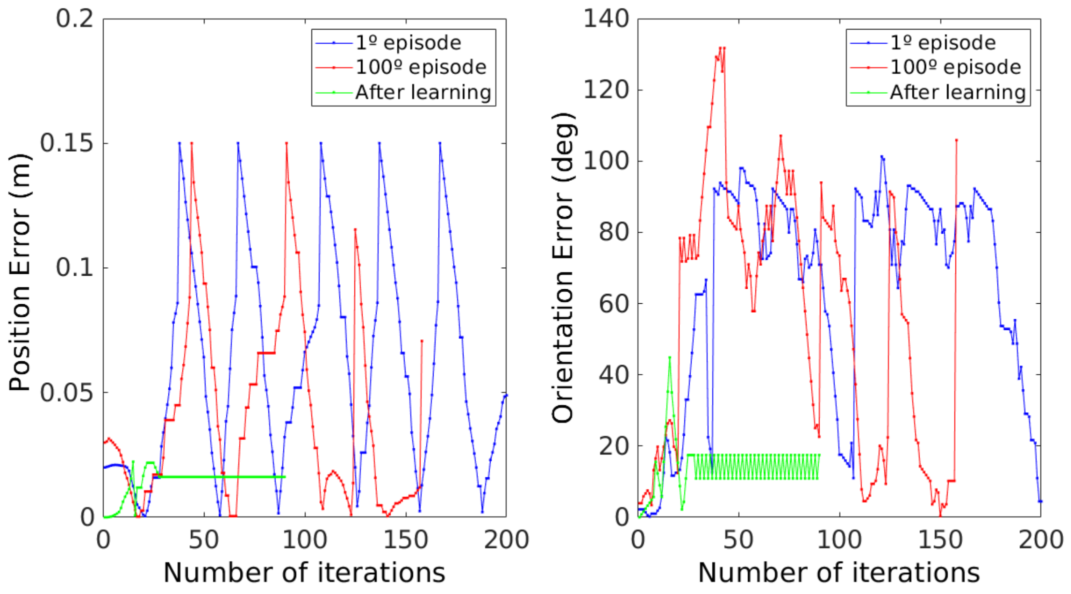


Figure 5.14: Error of the position and orientation of the robot with the automatic reward - Corner.

5.2 Simulations in "Doors"

For each of the maps, learning was performed using the e-greedy algorithm ($\epsilon = 0.1$) and annealing e-greedy ($\epsilon = \frac{1}{log((episode+1)^5)}$). The beginning of each episode starts with the robot in a random position at the entrance of the intersection and ends with the robot successfully completing the objective.

5.2.1 Right turn on a door

In order to consider as many states as possible, the algorithm using the e-greedy approach was first applied to the sub-task "Right turn on a door" using the first map with 50 episodes to determine if this number of episodes was enough and the number of states visited would have stabilized by then. Analyzing the Fig. 5.15 makes it possible to observe that after 50 episodes the robot still needs more learning time, because the number of states visited is not constant. Thus, the process was repeated for 100, 150, 200 and 300 episodes. In the same figure it is possible to analyze that the number of states visited start to stabilize before reaching 300 episodes showing that that number of episodes is sufficient for the learning process of each map. With the annealing e-greedy approach, the same strategy was used, and the same 300 episodes were sufficient, however, the experiment using the e-greedy approach allowed us to update more states (126) than the annealing e-greedy (122). Throughout the three maps of each sub-task, the Q-Learning table is reused, taking advantage of the update made in each of the experiments, since it is normal for some states to be revisited. It is possible to find out which ten states are the most visited during the learning in each map, which are probably also the states that the robot visits the most after learning (Fig. 5.16, C.1, C.2, C.3).

Then, it is also possible to analyze the error obtained between the trajectory performed by the robot and a straight line in the center of each intersection, and in the case of this sub-task, the reference trajectory makes an angle of 90 degrees in the middle of the intersection. A comparison between the error obtained in the first episode, in the hundredth episode and after learning shows the improvements throughout the learning process, where, in the first episode the number of iterations to reach the objective is much higher than after the learning is completed.

After the learning process was completed in each map, it is possible to use the Q-Learning table of each to simulate the robot's movement using the learning data. Therefore, it is possible to observe, in the Fig. 5.20, the path made by the robot in each of the maps previously described. As can be seen in the analysis of the images, it is possible to see that the learning process was successful, however, if the movement of the robot in a certain state is not the most correct one, the problem lies in the set of rewards given for this specific state.

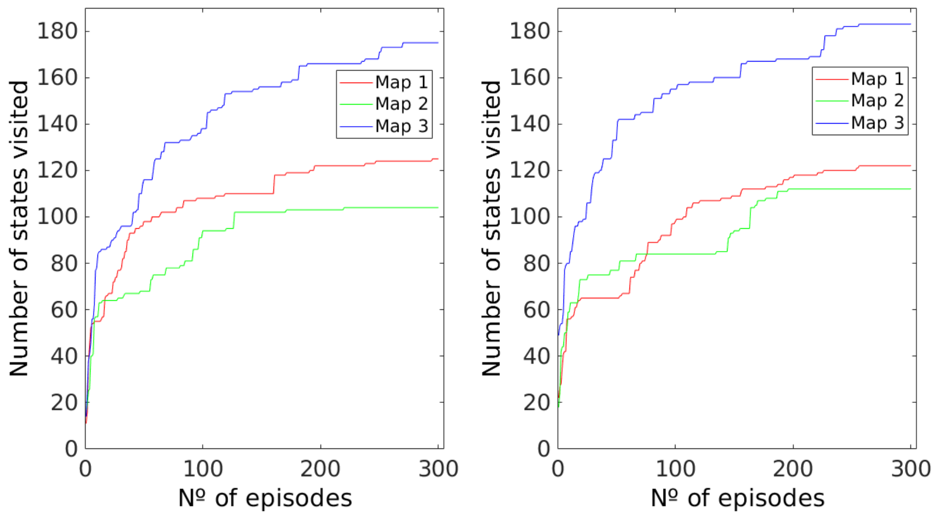


Figure 5.15: Sub-task "Right turn on a door" - states visited in function of the number of episodes.

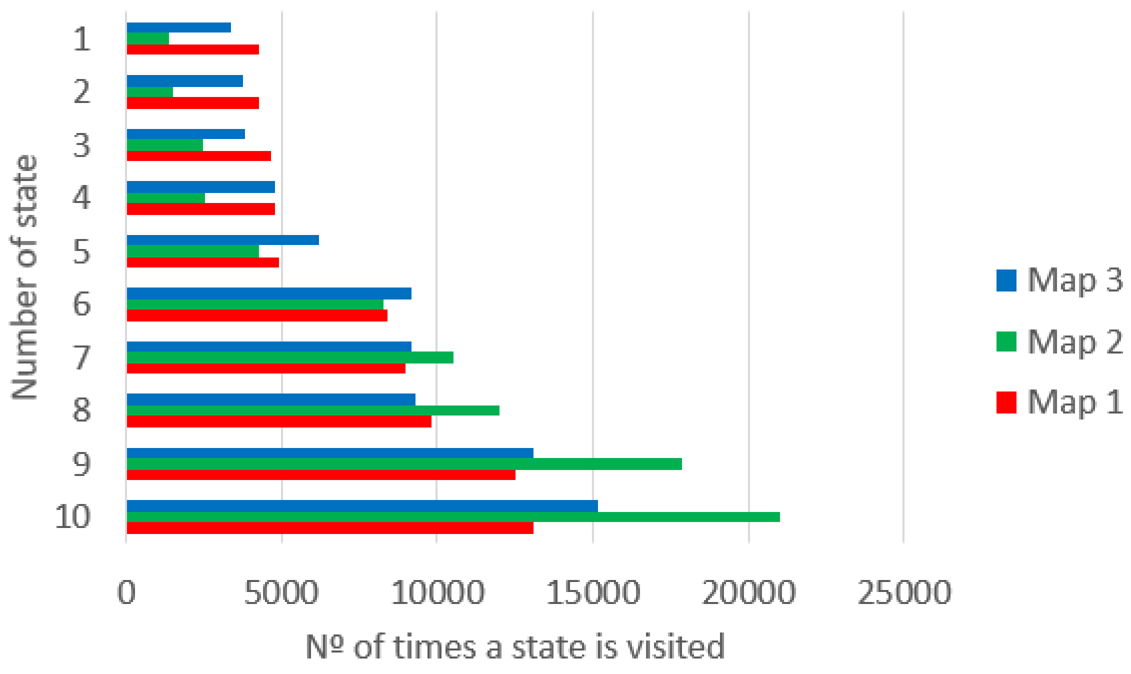


Figure 5.16: Number of times a state is visited - Right turn on a door.

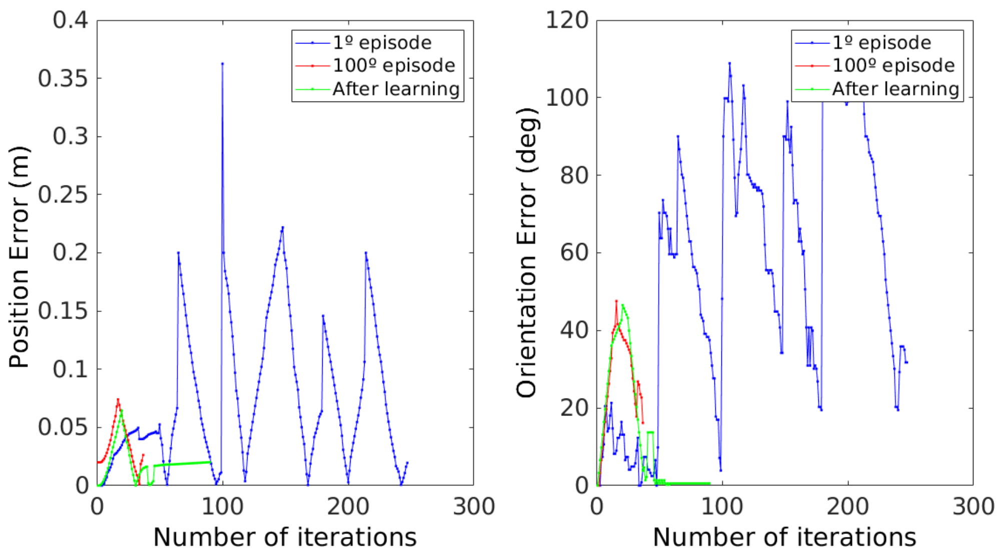


Figure 5.17: Error of the position and orientation of the robot - Map 1.

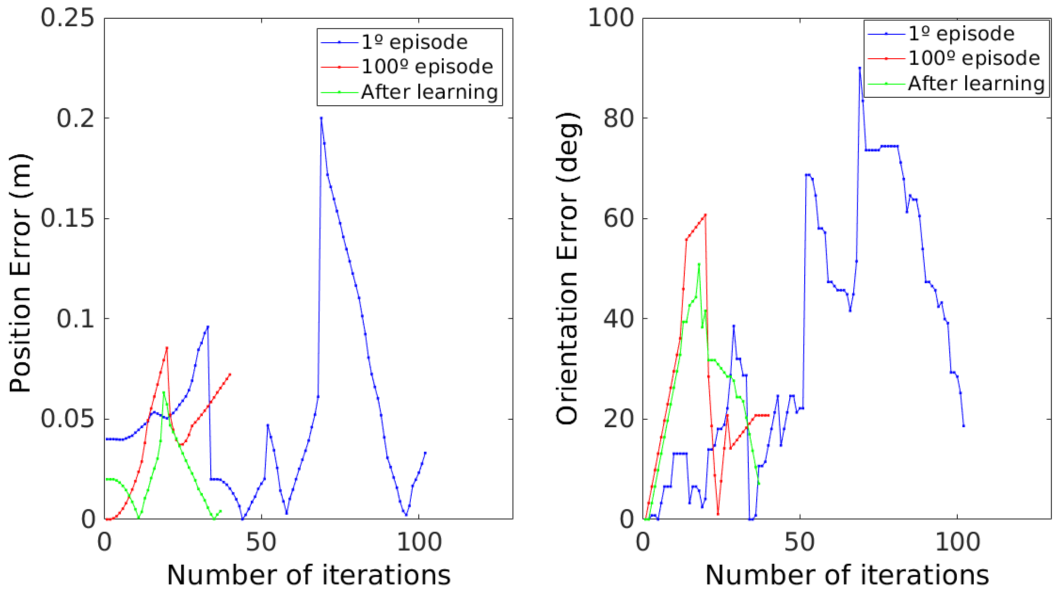


Figure 5.18: Error of the position and orientation of the robot - Map 2.

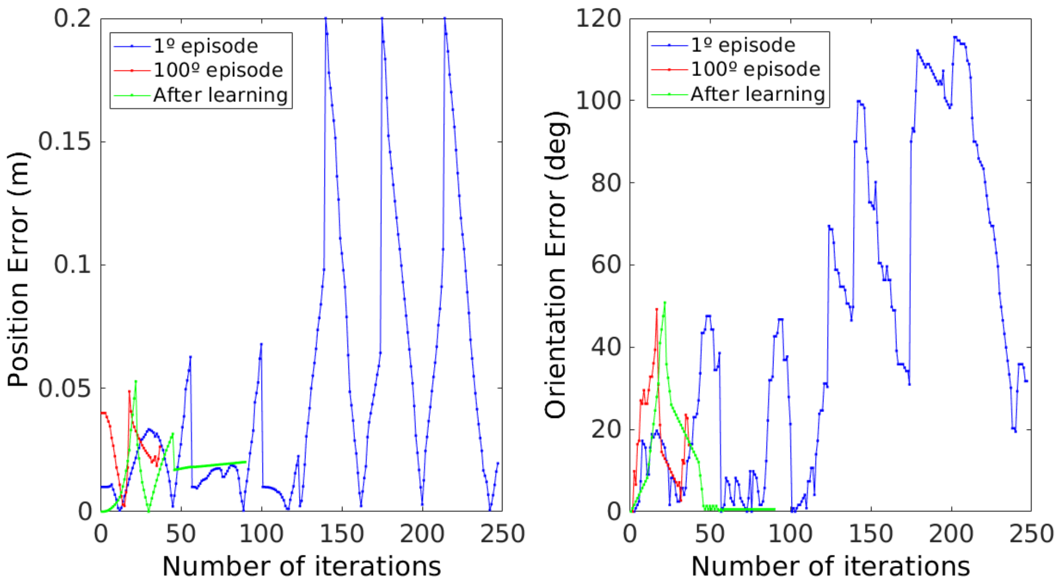


Figure 5.19: Error of the position and orientation of the robot - Map 3.

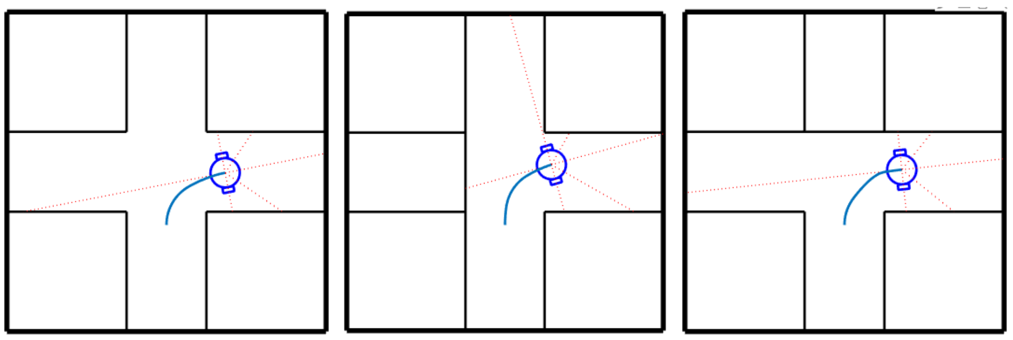


Figure 5.20: Trajectory of the robot using the updated Q-learning table - “Right turn on a door”.

5.2.2 Left turn on a door

All the tests performed in the ”Right turn on a door” sub-task were repeated for the ”Left turn on a door” sub-task and the results are very similar. The analysis of the number of states visited is shown in the figures 5.21, 5.22, C.4, C.5 and C.6. The graphs of the error of the position and orientation are presented in the figures 5.23, 5.24 and 5.25, and the trajectory made by the robot after the learning is showed in the Fig. 5.26. As can be seen in the analysis of the images, it is possible to see that the learning process was successful.

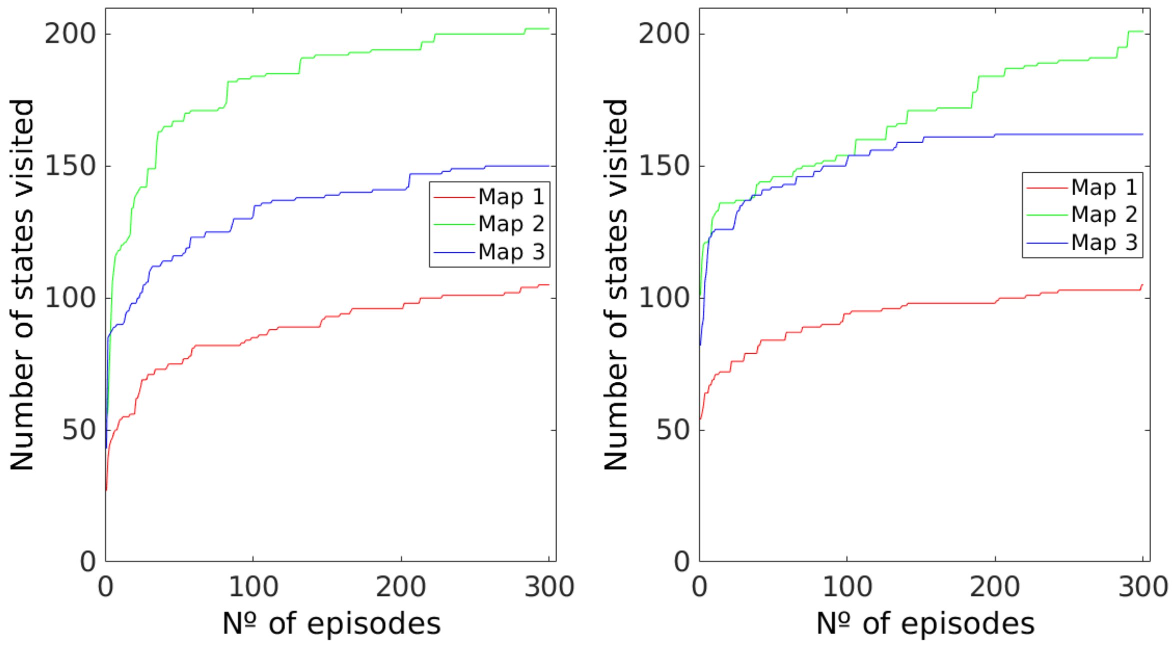


Figure 5.21: Sub-task “Left turn on a door” - states visited in function of the number of episodes - Map 1.

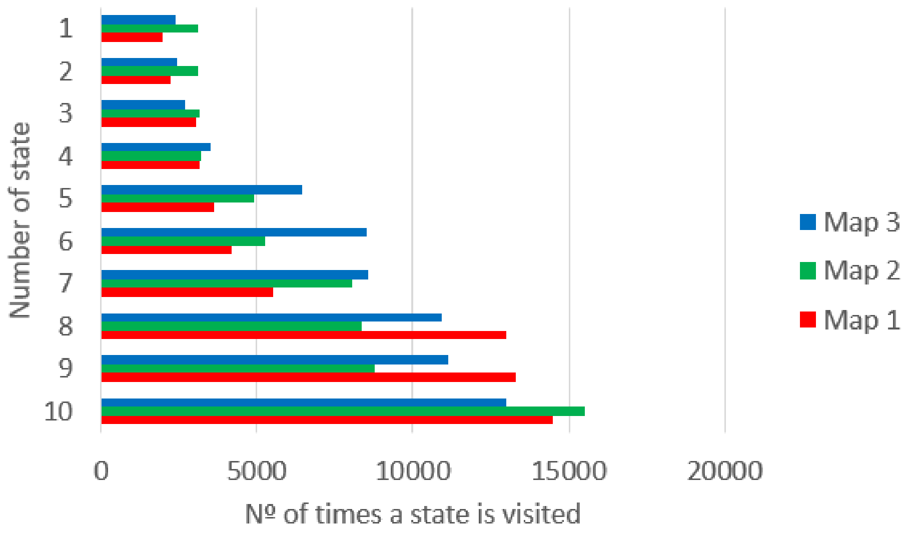


Figure 5.22: Number of times a state is visited - Left turn on a door - Map 1.

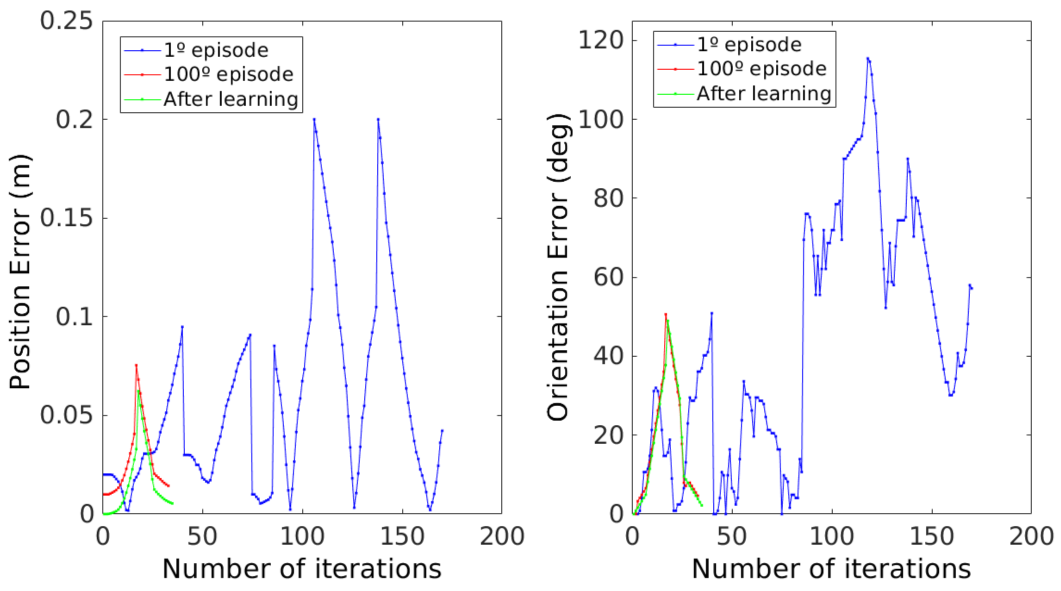


Figure 5.23: Error of the position and orientation of the robot - Map 1.

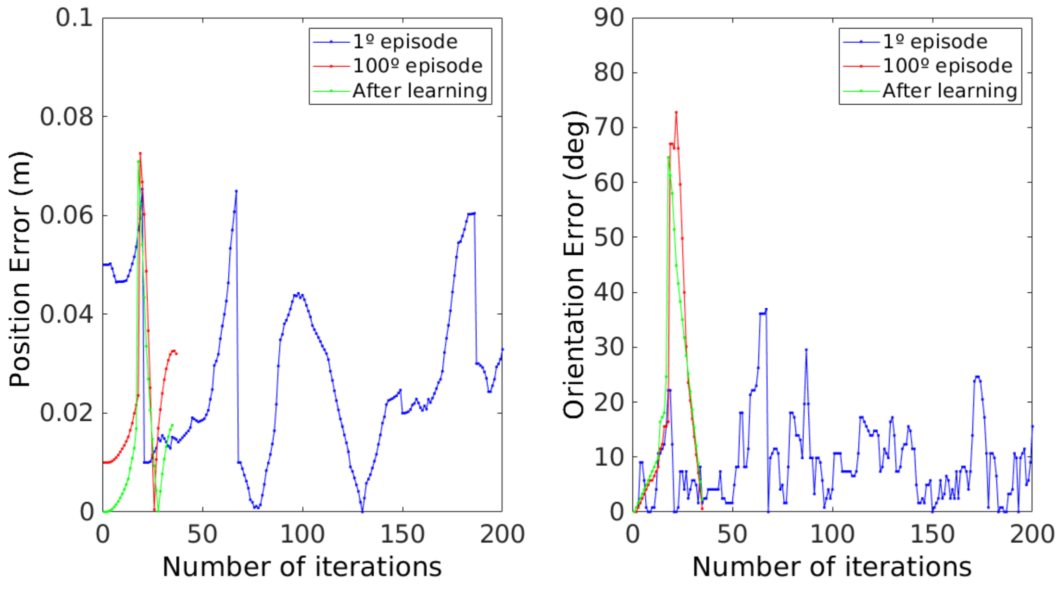


Figure 5.24: Error of the position and orientation of the robot - Map 2.

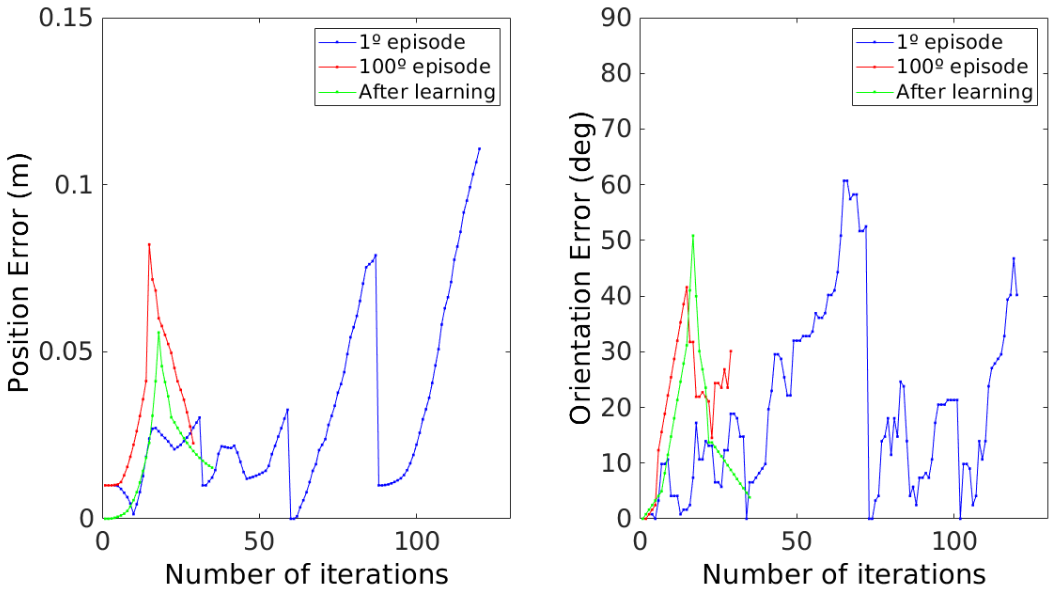


Figure 5.25: Error of the position and orientation of the robot - Map 3.

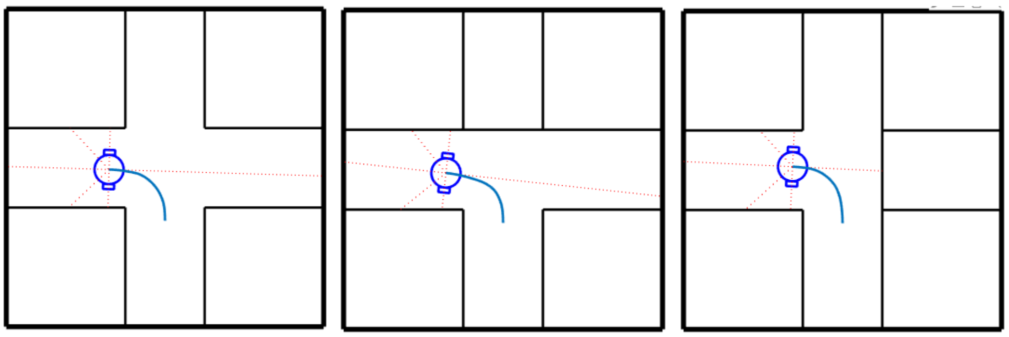


Figure 5.26: Trajectory of the robot using the updated Q-learning table -"Left turn on a door".

5.2.3 Going forward on a door

All the tests performed in the "Right turn on a door" and "Left turn on a door" sub-tasks were repeated for the "Going forward on a door" sub-task and the results are again very similar. The analysis of the number of states visited is shown in the figures 5.27, 5.28, C.7, C.8, C.9. The graphs of the error of the position and orientation are presented in the figures 5.29, 5.30 and 5.31, and the trajectory made by the robot after the learning is showed in the figure 5.32. When analyzing all the information previously presented it is possible to realize that the robot learns to behave quickly and smoothly when it arrives at a door, always managing to complete its goal, which is to return to a corridor.

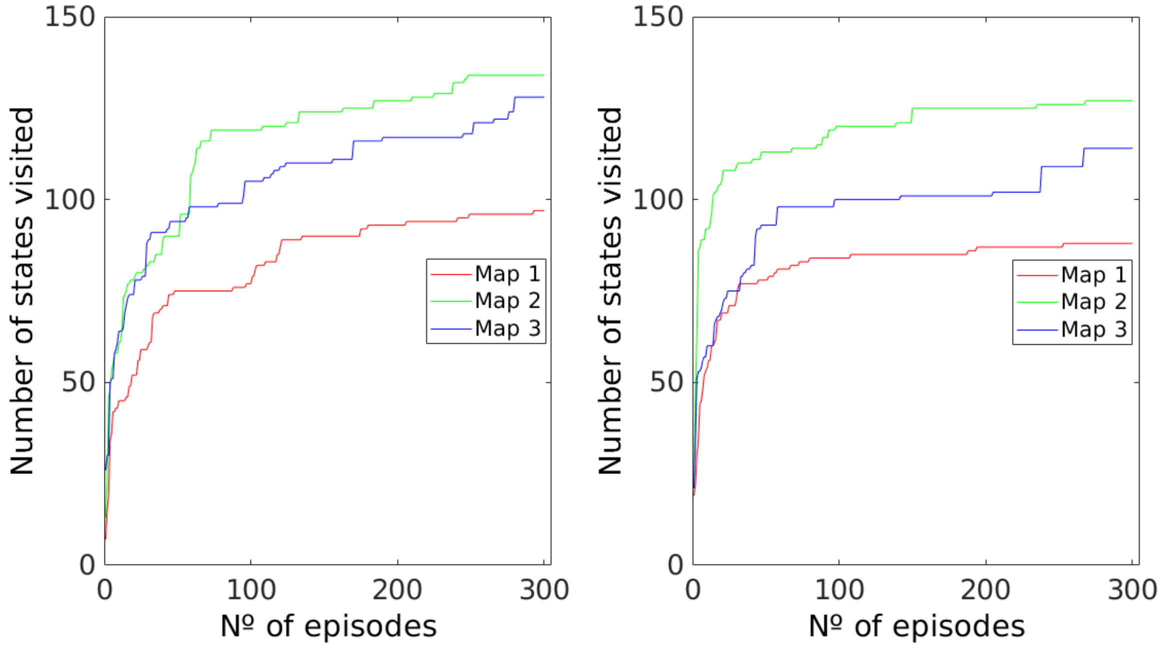


Figure 5.27: Sub-task "Going forward on a door" - states visited in function of the number of episodes - Map 1.

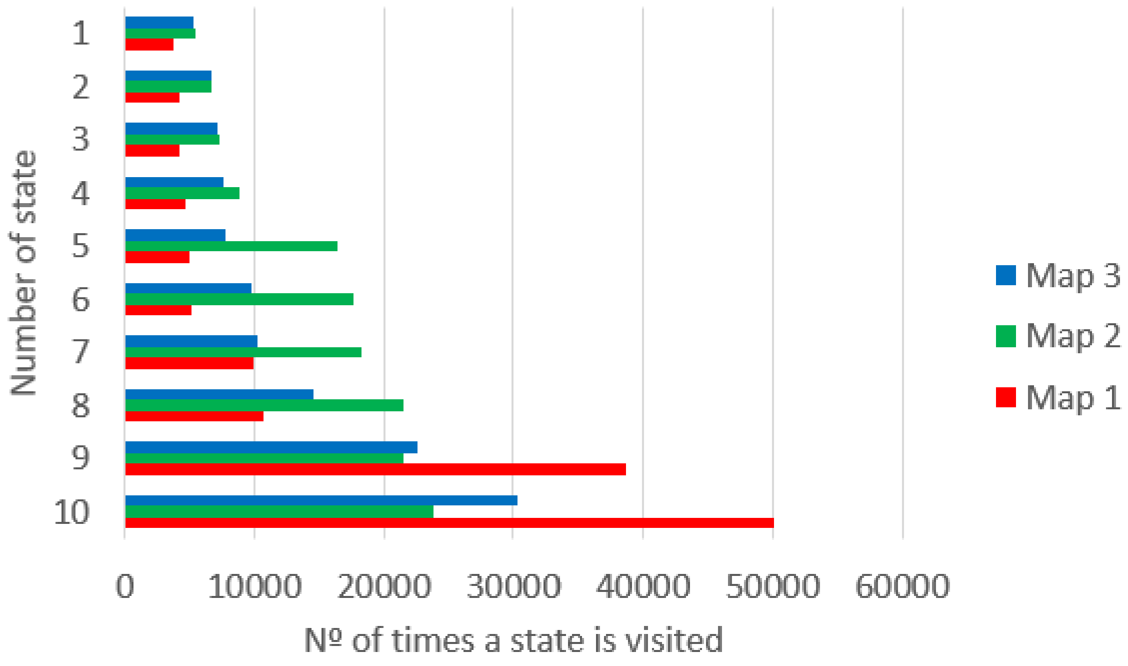


Figure 5.28: Number of times a state is visited - Going forward on a door - Map 1.

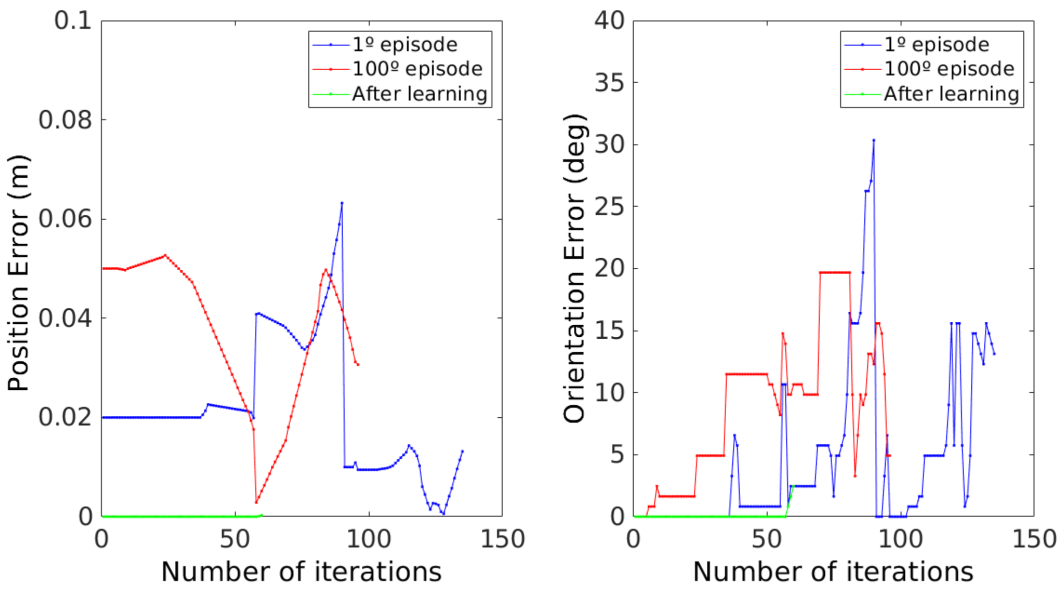


Figure 5.29: Error of the orientation and position of the robot - Map 1.

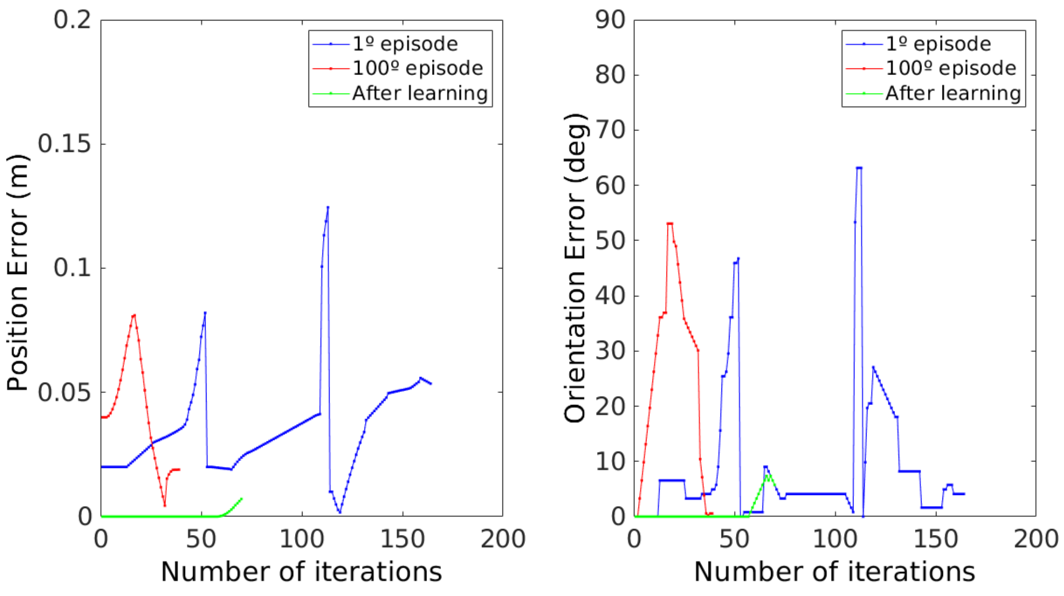


Figure 5.30: Error of the orientation and position of the robot - Map 2.

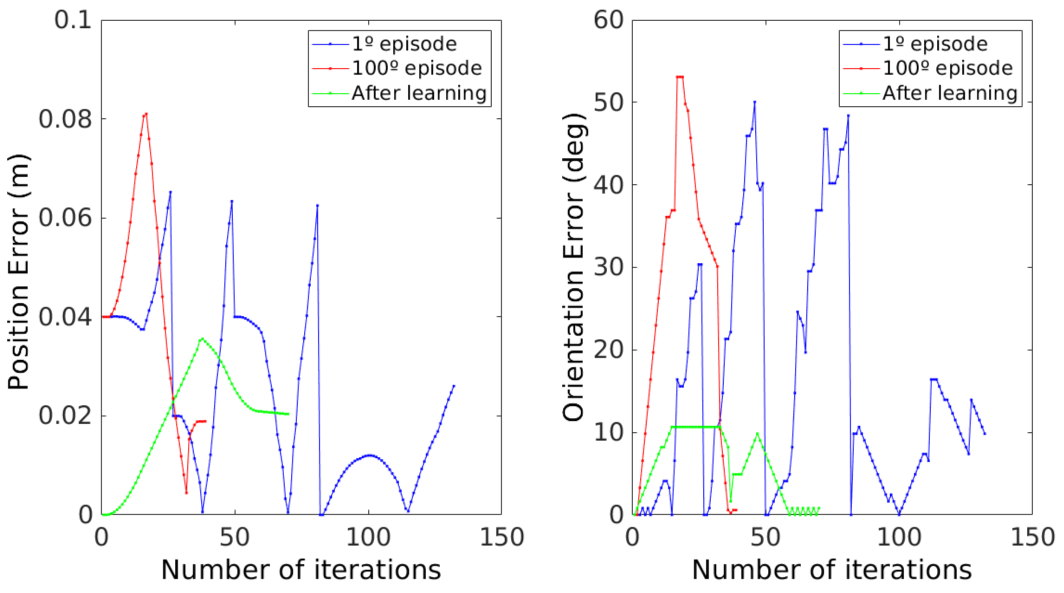


Figure 5.31: Error of the orientation and position of the robot - Map 3.

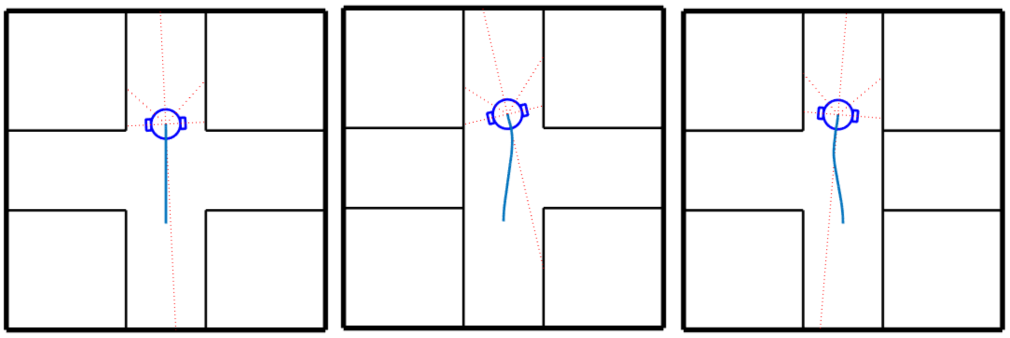


Figure 5.32: Trajectory of the robot using the updated Q-learning table - ”Going forward on a door”.

5.3 Simulation of the Higher Level

The first method does not necessarily give the robot the fastest route to the goal, because the reward function does not give the information about the iterations needed for the robot to reach the end of a sub-task so as can be seen in the Table 5.2:

State\Option	o_1 (Go Forward)	o_2 (Turn right)	o_3 (Turn left)
s_1	-0.0891	4.1091	-0.2143
s_2	3.4693	-0.1791	-0.2999
s_3	-0.2453	-0.4057	2.7613
s_4	0.4492	-0.1889	-2.7100
s_5	0.4379	-2.7100	-0.2243
s_6	-1.9000	0.1441	-0.2104
s_7	0	0	0

Table 5.2: Q-Learning table updated after the learning process using the first method.

For example, if the robot reaches the checkpoint 4, the option that would take him faster to the goal would be turning right, however, the algorithm at the end of the learning process selected the option “Go forward” (Fig. 5.33).

The second method proved to reach good results because the reward function depends on the number of iterations needed to reach the goal. The Q-Learning table after learning is presented below, Table 5.3:

State\Option	o_1 (Go Forward)	o_2 (Turn right)	o_3 (Turn left)
s_1	-0.8022	4.2214	-2.8231
s_2	1.0362	-0.2619	-1.2581
s_3	-0.9087	-1.4981	2.9764
s_4	-1.5685	0.6021	-3.4380
s_5	-0.8641	-5.2170	0.6751
s_6	-1	-0.5960	-0.3927
s_7	0	0	0

Table 5.3: Q-Learning table updated after the learning process.

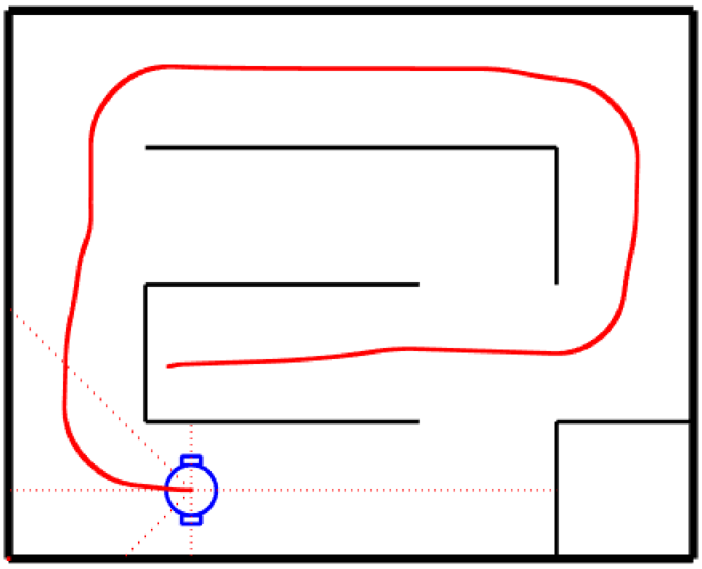


Figure 5.33: Trajectory chosen not favoring the fastest path.

In the figure 5.34 it is possible to observe that the robot, in the beginning of the learning process, can not find the goal right away which is normal, considering it does not have any information about the maze and how to reach the goal. Figure 5.35 shows the behavior of the agent, after having completed the learning process, where it can be verified that the agent chooses the fastest way to reach the goal.

In order to understand the effect that the Q-Learning algorithm produces during learning, it is possible to analyze the figure 5.36, where the number of iterations and the reward sum as a function of the number of the episode is presented. As the number of episodes increases, the robot finds the way to the goal much faster, except for rare exceptions in which the method of selecting the option to be taken, be it e-greedy or annealing e-greedy, chooses an action that takes it away from the goal.

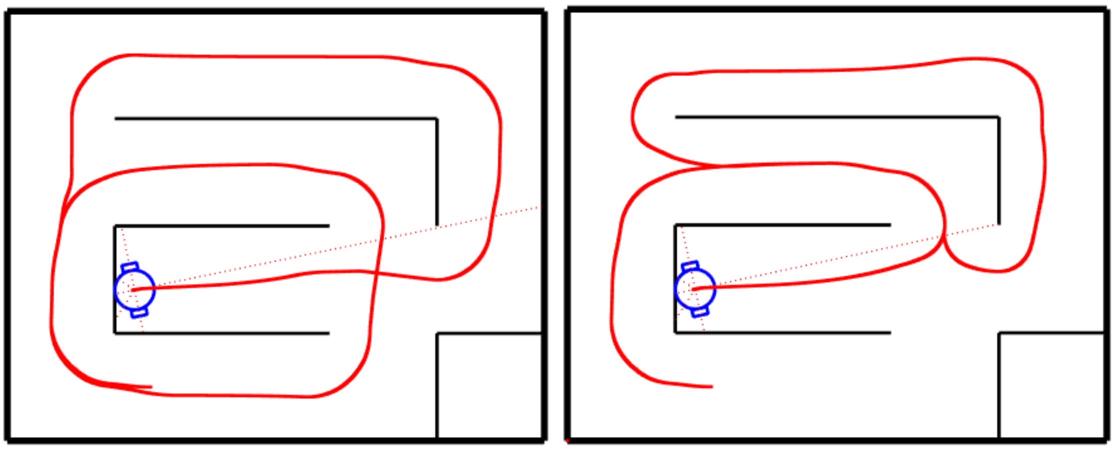


Figure 5.34: First and second episodes of the Hierarchical Reinforcement Learning process.

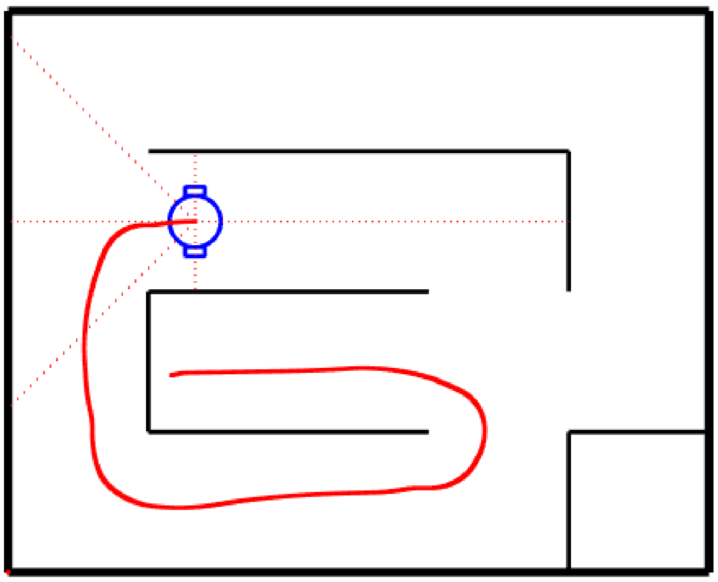


Figure 5.35: Robot behavior after the Hierarchical Reinforcement Learning process.

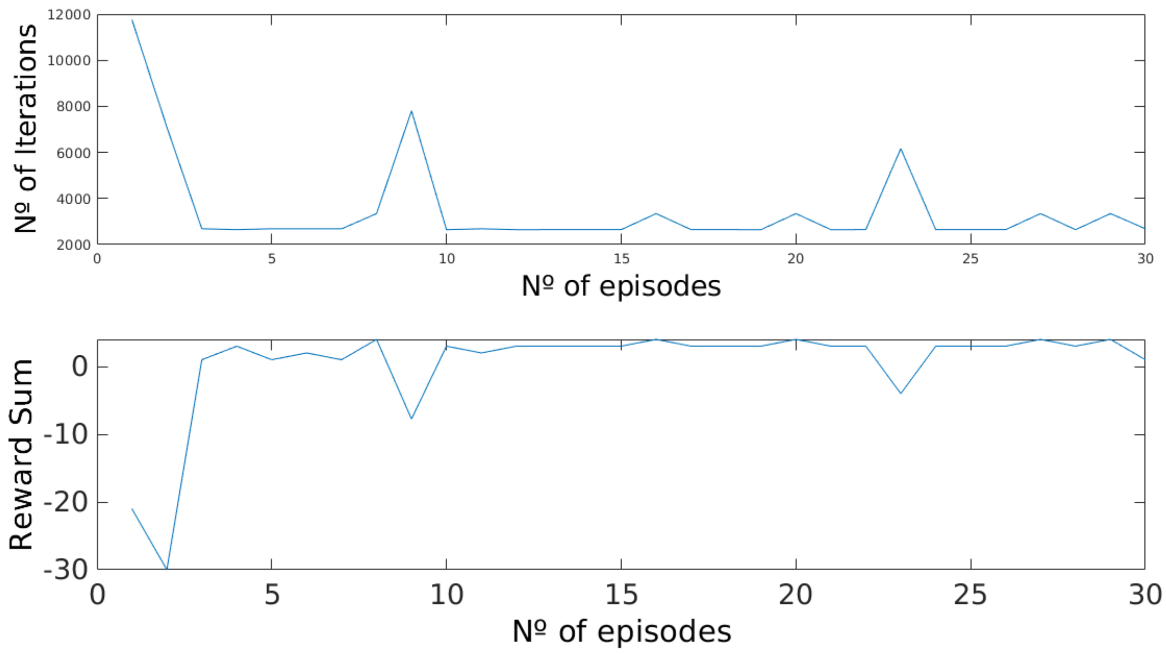


Figure 5.36: Number of iterations and reward sum as a function of the number of episodes.

The method that uses the state machine gives the same results as the second method, however, it converges way faster because it is not necessary to go through any learning in the maze, which is what consumes the most time throughout the simulation, as the passage between sub-tasks is not instantaneous. The Q-Learning Table obtained is presented in 5.4 where the option with the higher value in each of the states is the same as the one in the Table 5.3 showing that the robot always opts for the fastest route to reach the goal.

State\Option	o_1 (Go Forward)	o_2 (Turn right)	o_3 (Turn left)
s_1	-0.0562	5.9216	0.2812
s_2	3.1497	-0.19	-0.1106
s_3	-0.0195	0.0508	3.0011
s_4	-0.0144	2.8626	-5.2170
s_5	0.1354	-5.2170	3.1462
s_6	-2.7100	-0.3094	-0.1000
s_7	0	0	0

Table 5.4: Q-Learning table updated after the learning process using the state machine approach.

If the table is completely updated, it is possible to join the two levels, the high level and the low level. For test purposes, by placing the robot at a random position on the full maze, it is possible to visualize that it finds the fastest route to reach the goal, going through each of the sub-tasks using the corresponding Q-Learning table of each one (Fig. 5.37). This shows that the division of the maze into sub-tasks allows an easier and quicker learning than approaching the maze directly.

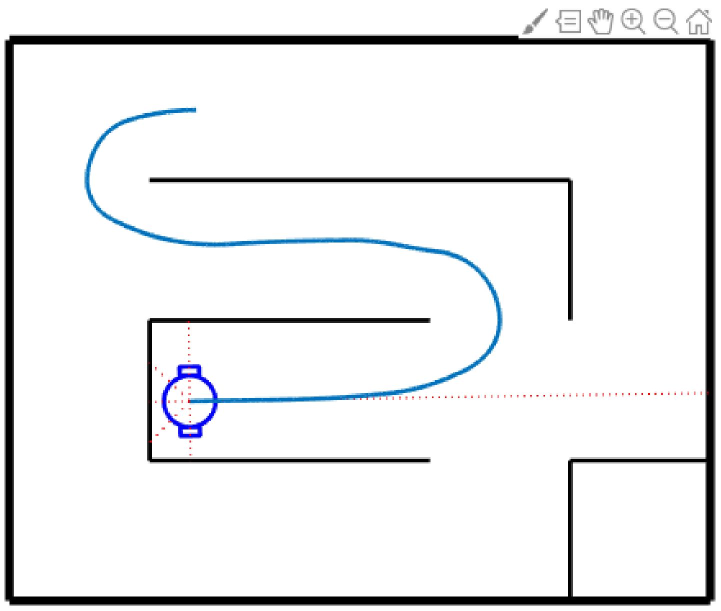


Figure 5.37: Trajectory made by the robot in the complete maze.

5.4 Experiments in the Real Environment

The approach to the real environment was first done without any prior learning in simulation, to see if it would be possible for the robot to learn in a short time to move in any of the sub-tasks, however, this process was quickly discarded due to its slowness, as the robot was constantly hitting the wall in the initial phase, having to be moved by hand every time that happened.

So, the second option was to use transfer learning, passing the learning performed in simulation directly to the real robot. This experiment used the data obtained using the first version of the reward function created with the support of the user. When it was visible that the robot behaved in a similar way to the behavior evidenced in simulation, it was understood that the transfer learning method proved to be extremely viable. In the figures 5.38, 5.39 and 5.40 it is possible to observe the trajectory used by the robot beginning at random points of the maze and using the information obtained during the learning performed in simulation. As it can be seen, the robot already recognizes that it is at a checkpoint when it finds a bar code on the floor. Taking into account the table 5.3, the robot decides correctly which of the sub-tasks it must choose in each moment, reaching the goal using the fastest route. The main disadvantage of this technique is the fact that it can not solve situations where the robot is very near the walls due to the nonexistence of actions to rotate on its own axis.

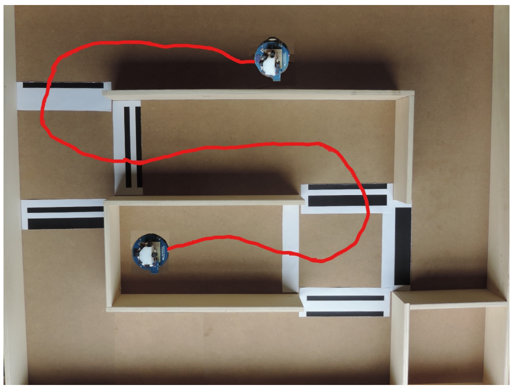


Figure 5.38: First trajectory made by the robot in the complete real environment.

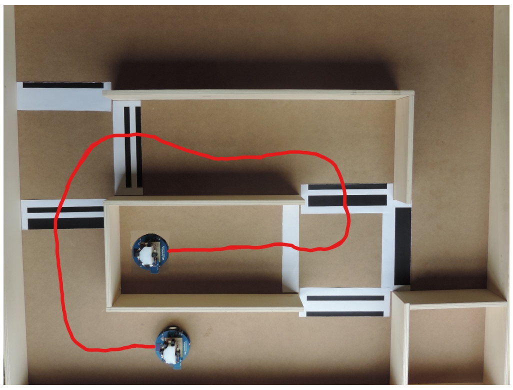


Figure 5.39: Second trajectory made by the robot in the complete real environment.

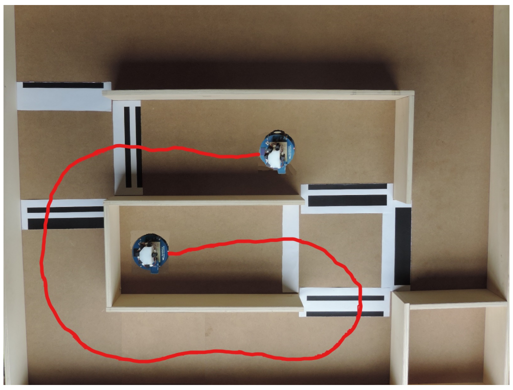


Figure 5.40: Third trajectory made by the robot in the complete real environment.

Next, the same experiment was performed using the Q values table in the “Corridors and Corners” sub-task obtained through the automatic reward function explained in the previous chapter. It should be kept in mind that in this case, the number of actions is higher, 11 actions, than the number of actions previously used in this sub-task, 9 actions. However, the remaining sub-tasks related to entering a port maintained the same learning tables. When analyzing the trajectory of the robot starting from random positions of the maze is again possible to confirm that it can reach the goal using the fastest possible route (Figs. 5.41, 5.42). This technique allows the robot never to hit the walls thanks to the two new actions of rotating around itself, whereas in the previous case, if the robot ever hit a wall, it had no possible action to get out of this situation. However, the movement of the robot in the corners is not very smooth and sometimes the robot gets stuck between two states where the action with the maximum Q value is to rotate to the left and to the right. This shows that the reward function created for the corner needs to be thought in the future to give better results. The main disadvantage of this technique is that if the action chosen by the robot on the checkpoint is to rotate on its own axis, causing the reading of the bar code to fail, leading to a wrong processing where another checkpoint can be read, or even a nonexistent one.

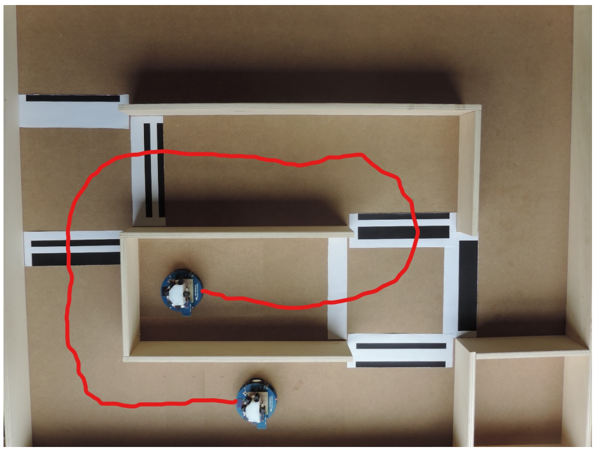


Figure 5.41: First trajectory made by the robot in the complete real environment using the automatic reward function.

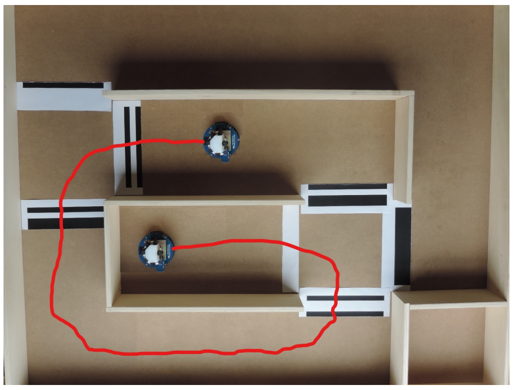


Figure 5.42: Second trajectory made by the robot in the complete real environment using the automatic reward function.

As the results obtained using only transfer learning were positive, the last task to be carried out was an online learning based on the learning tables previously obtained in simulation. As expected, the robot maintained the same behavior showing that it is possible to use transfer learning in a problem of this sort, continuing the learning in the real environment.

Chapter 6

Conclusions and Future Work

This chapter summarizes the major contributions of this dissertation and discusses possible developments for future work.

6.1 Conclusions and Contributions

This dissertation considered the navigation problem of a small-size omnidirectional mobile robot in a maze-like environment. A HRL framework was implemented within a kinematics simulation and visualization tool in order to validate the algorithms prior to implementation in an Arduino-based physical platform. In order to define the robots state, the RL algorithm uses six distance measurements supplied by infrared sensors that detect the surrounding maze walls. Extensive experimental validation proves the validity and robustness of the hierarchical approach and demonstrates the important role of the reward design.

The main contributions and limitations of this work can be summarized as follows:

- The robotic infrastructure proved to be adequate to the work, at the hardware and software level. However, the communication module did not always respond correctly throughout the experiments, having to be restarted occasionally, and the working frequency could also be somewhat higher. The ROS architecture, despite being a very powerful tool, was not used exhaustively because it was not part of the previous knowledge obtained during the academic course, having served its purpose in an effective way.

- The HRL approach introduced in this dissertation integrates ideas from the most well-known frameworks. The hierarchical structure allowed a faster learning of a policy made up of two levels so that the mobile robot can select to perform sequences of lower-level actions. The results achieved, in a simulated and in a real robot, demonstrate the effectiveness of the learning approach for mobile navigation in maze-like environments. In this case, HRL proved to be a valid alternative to overcome some weaknesses of classical RL, such as data and learning inefficiency. If the maze under study has a different layout of the maze used during the experiments, first it is necessary to create a topological map with the help of the checkpoints and to perform a high-level learning so that the robot learns the best options to take in every moment of decision. If the measurements of the corridors and doors are different from the ones used in this dissertation, the transfer of the simulation results to the real environment may not be direct and some adjustments must be made in simulation. Throughout the experiments, objects were never placed along the maze to know if the robot was able to overcome them, however, it is expected that as the robot moves correctly down the maze it is also capable of not colliding with any object.

- The experimental validation of the work in which the real mobile robot RLAN-bot navigate through a maze to reach a goal is another contribution of this dissertation. The ease with which the simulation results were directly transferred to the robot was, perhaps, the most surprising aspect of the work, particularly because the simulated model solely considers the kinematics of the task. Although some additional work is required, this shows that different levels of hierarchy can involve different knowledge, allowing for better transfer.

6.2 Future Work

There is a lot of work that can be improved since this topic is a very extensive and always evolving case study. The perspectives of future work include both improvements to the work done and new directions of investigation:

- To improve the communications between the mobile robot and the host computer to increase the current baud rate and frequency.
- To consider the inclusion of a greater number of IR-sensors and their fusion with other sensorial modalities.
- To compare the performance of the HRL approach with other formulations to evaluate its added value in the context of robot navigation.
- To evaluate the advantages of other algorithms beyond Q-learning and the impact of the hyper-parameters will also require additional efforts.
- Directions of future research can include the evaluation of the generalization capabilities of the HRL approach (facing dynamic obstacles for example), the study of alternative methods to design and/or learn the reward function (using IRL algorithms for example), the integration of hierarchical and model learning methods and the exploration of continuous state-action spaces through function approximation.

References

[1] Stuart J Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Prentice Hall - Artificial Intelligence A Modern Approach*. 2001.

[2] S. Barry Cooper and Jan Van Leeuwen. Computing Machinery and Intelligence. *Alan Turing: His Work and Impact*, pages 551–621, 2013.

[3] Thomas M. Mitchell. *Machine Learning*. 2003.

[4] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[5] L.Fridman et al. Human Side of Tesla Autopilot: Exploration of Functional Vigilance in Real-World Human-Machine Collaboration . *Center for Transportation and Logistics, Massachusetts Institute of Technology (MIT)*, 2019.

[6] E. Rosten and T. Drummond. Machine learning for highspeed corner detection. *Computer VisionECCV. Springer Berlin Heidelberg*, pages 430–443, 2006.

[7] A. Giusti et al. A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters*, pages 661–667, 2016.

[8] L. Wang B. Zuo, J. Chen and Y. Wang. A reinforcement learning based robotic navigation system. *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3452–3457, 2014.

[9] X. Jiao X. Song, H. Fang and Y. Wang. Autonomous mobile robot navigation using machine learning. *2012 IEEE 6th International Conference on Information and Automation for Sustainability*, pages 135–140, 2012.

[10] H. Kim D. Baek, M. Hwang and D. Kwon. Path Planning for Automation of Surgery Robot based on Probabilistic Roadmap and Reinforcement Learning. *018 15th International Conference on Ubiquitous Robots (UR)*, pages 342–347, 2018.

[11] John J. Leonard and Hugh F. Durrant-Whyte. Mobile Robot Localization by Tracking Geometric Beacons. *IEEE Transactions on Robotics and Automation*, 7(3):376–382, 1991.

[12] MTGR - The Worlds Lightest Full-Featured Tactical Ground Robot. `http://www.robo-team.com/products/mtgr/`. Accessed in 2018-04-07.

[13] TALON Small Mobile Robot. `https://www.globalsecurity.org/military/systems/ground/talon.htm`. Accessed in 2018-04-07.

[14] Predator RQ-1 / MQ-1 / MQ-9 Reaper UAV. `https://www.airforce-technology.com/projects/predator-uav/`. Accessed in 2018-04-07.

[15] Prime Air. `https://www.aboutamazon.co.uk/innovation/prime-air`. Accessed in 2018-04-07.

[16] PLUTO PLUS, PLUTO and PLUTO-L - medium size minehunting ROVs. `http://www.idrobotica.com/pluto-plus.php`. Accessed in 2018-04-07.

[17] SeaBed. `https://www.whoi.edu/what-we-do/explore/underwater-vehicles/auvs/seabed/`. Accessed in 2018-04-07.

[18] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[19] Richard Bellman. *Dynamic Programming.* Dover Publications, 1957.

[20] Christopher Watkins. *Learning From Delayed Rewards.* PhD thesis, Kings College, 1989.

[21] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[22] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University, 1994.

[23] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. pages 1038–1044, 1996.

[24] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181 – 211, 1999.

[25] Andrea Buitrago, Fernando De la Rosa, and Fernando Lozano-Martinez. Hierarchical reinforcement learning approach for motion planning in mobile robotics. 10 2013.

[26] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, pages 1043–1049, Cambridge, MA, USA, 1998. MIT Press.

[27] Thomas G. Dietterich. An overview of maxq hierarchical reinforcement learning. In Berthe Y. Choueiry and Toby Walsh, editors, *Abstraction, Reformulation, and Approximation*, pages 26–44, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[28] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Int. Res.*, 13(1):227–303, November 2000.

[29] Jing Shen, Ochang Gu, and Haibo Liu. Multi-agent hierarchical reinforcement learning by integrating options into MAXQ. *First International Multi- Symposiums on Computer and Computational Sciences, IMSCCS'06*, 1:676–682, 2006.

[30] V. Madhu Babu, U. Vamshi Krishna, and S. K. Shahensha. An autonomous path finding robot using Q-learning. *Proceedings of the 10th International Conference on Intelligent Systems and Control, ISCO 2016*, 1:1–6, 2016.

[31] Bashan Zuo, Jiaxin Chen, Larry Wang, and Ying Wang. A reinforcement learning based robotic navigation system. *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, 2014-Janua(January):3452–3457, 2014.

[32] Daniel Paul Romero-Marti, Jose Ignacio Nunez-Varela, Carlos Soubervielle-Montalvo, and Alfredo Orozco-De-La-Paz. Navigation and path planning using reinforcement learning for a Roomba robot. *18th Congreso Mexicano de Robotica, COMRob 2016*, 2017.

[33] Jawad Muhammad and Ihsan Omur Bucak. An improved Q-learning algorithm for an autonomous mobile robot navigation problem. *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering, TAEECE 2013*, (I):239–243, 2013.

[34] Toms Martínez-Marín. On-line optimal motion planning for nonholonomic mobile robots. *Proceedings - IEEE International Conference on Robotics and Automation*, 2006(May):512–517, 2006.

[35] Tomas Martinez-Marin and Rafael Rodriguez. Navigation of Autonomous Vehicles in Unknown Environments using Reinforcement Learning. *2007 IEEE Intelligent Vehicles Symposium*, (3):872–876, 2007.

[36] Toms Martínez-Marín and Tom Duckett. Fast reinforcement learning for vision-guided mobile robots. *Proceedings - IEEE International Conference on Robotics and Automation*, 2005(April):4170–4175, 2005.

[37] B Bakker, V Zhumatiy, G Gruener, and J Schmidhuber. Quasi-online Reinforcement Learning for Robots. (TR IDSIA-05-03?):2997–3002, 2005.

[38] B R Leffler, M L Littman, and T Edmunds. Efficient reinforcement learning with relocatable action models. *Proceedings of the National Conference on Artificial Intelligence*, 1:572–577, 2007.

[39] Ronen Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. volume 3, pages 953–958, 01 2001.

[40] c't Bot and c't Sim - The free and open source robotic project. `https://www.ct-bot.de/`. Accessed in 2018-12-15.

[41] KHEPERA IV NEW K-Team Corporation. `https://www.k-team.com/khepera-iv`. Accessed in 2018-12-10.

[42] AlphaBot2-Ar - Waveshare Wiki. `https://www.waveshare.com/wiki/AlphaBot2-Ar#Documentation`. Accessed in 2018-10-8.

[43] Waveshare. *AlphaBot 2 User Manual*, 8 2017. V1.3.

[44] Arduino Uno Plus. `https://www.waveshare.com/UNO-PLUS.htm`. Accessed in 2018-10-9.

[45] Everlight. *Opto Interrupter ITR20001/T*, 11 2016. Rev.7.

[46] ElecFreaks. *Ultrasonic Ranging Module HC-SR04*.

[47] P. Castillo-Pizarro, T. V. Arredondo, and M. Torres-Torriti. Introductory survey to open-source mobile robot simulation software. In *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, pages 150–155, Oct 2010.

[48] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and performance comparison of the v-rep, gazebo and argos robot simulators. 02 2018.

[49] V-REP. `http://www.coppeliarobotics.com/`. Accessed in 2018-11-21.

[50] Swarmanoid Project. `http://www.swarmanoid.org/`. Accessed in 2018-11-21.

[51] ARGoS. `https://www.argos-sim.info/`. Accessed in 2018-11-21.

[52] Gazebo. `http://gazebosim.org/`. Accessed in 2018-11-21.

[53] Pololu. *SHARP GP2Y0A41SK0F - Distance measure sensor unit.*

[54] John S. Bridle. Training Stochastic Model Recognition Algorithms as Networks can lead to Maximum Mutual Information Estimation of Parameters . *Advances in Neural Information Processing Systems 2*, pages 211–217, 1990.

[55] Arryon Tijsma, Madalina Drugan, and Marco Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. 2016.

[56] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

[57] Stuart Russell. Learning agents for uncertain environments (extended abstract). In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 101–103, New York, NY, USA, 1998. ACM.

[58] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *In Proceedings of the Twenty-first International Conference on Machine Learning*. ACM Press, 2004.

[59] Sergey Levine, Zoran Popovic, and Vladlen Koltun. Nonlinear inverse reinforcement learning with gaussian processes. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 19–27. Curran Associates, Inc., 2011.

Appendix A

Schematic of the Alphabot2-AR

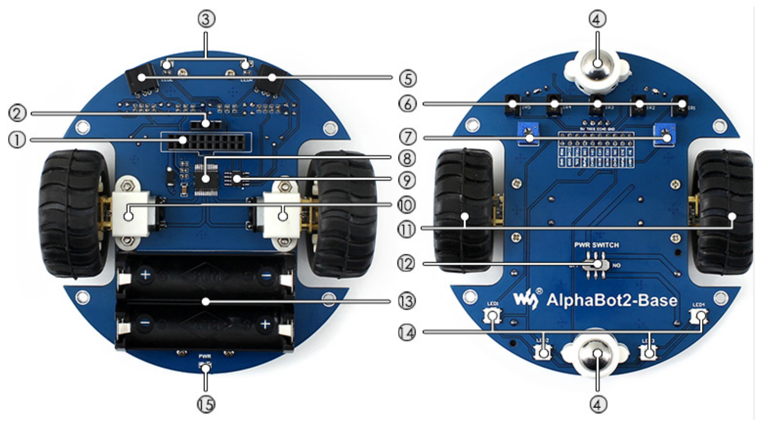


Figure A.1: AlphaBot2-Base Schematic.

1. Ultrasonic module interface
2. AlphaBot2 control interface
3. Obstacle avoidance indicators
4. Omni-direction wheel
5. Reflective infrared photoelectric sensor, ST188, for obstacle avoidance
6. Reflective infrared photoelectric sensor, ITR20001, for line tracking
7. Potenciometer for adjusting obstacle avoiding range
8. Dual H-bridge motor drive,TB6612FNG
9. Voltage comparator, LM393
10. N20 micro gear motor reduction rate 1:30 6V/600RPM
11. Rubber wheels, with a diameter of 42mm, and 19mm width

12. Power switch
13. Battery holder, for 14500 batteries
14. WS2812B: LEDs RGB
15. Power Indicator

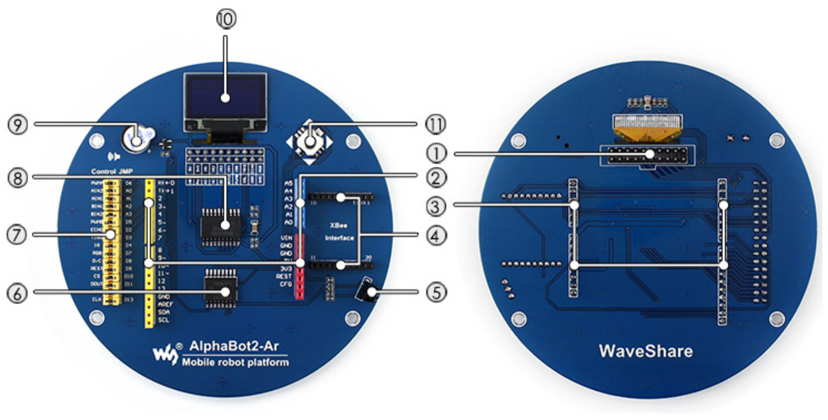


Figure A.2: AlphaBot2-Ar Schematic.

1. AlphaBot2 control interface to connect to AlphaBot2-Base
2. Arduino expansion header
3. Arduino interface to connect a compatible Arduino controller
4. XBee connector
5. IR receiver
6. PC8574: I/O expander, SPI interface
7. Arduino peripheral jumpers
8. TLC1543: 10-bit AD acquisition chip
9. Buzzer
10. 0.96inch OLED SSD1306 driver, 128x64 resolution
11. Joystick

Appendix B

Final Version of the PCB

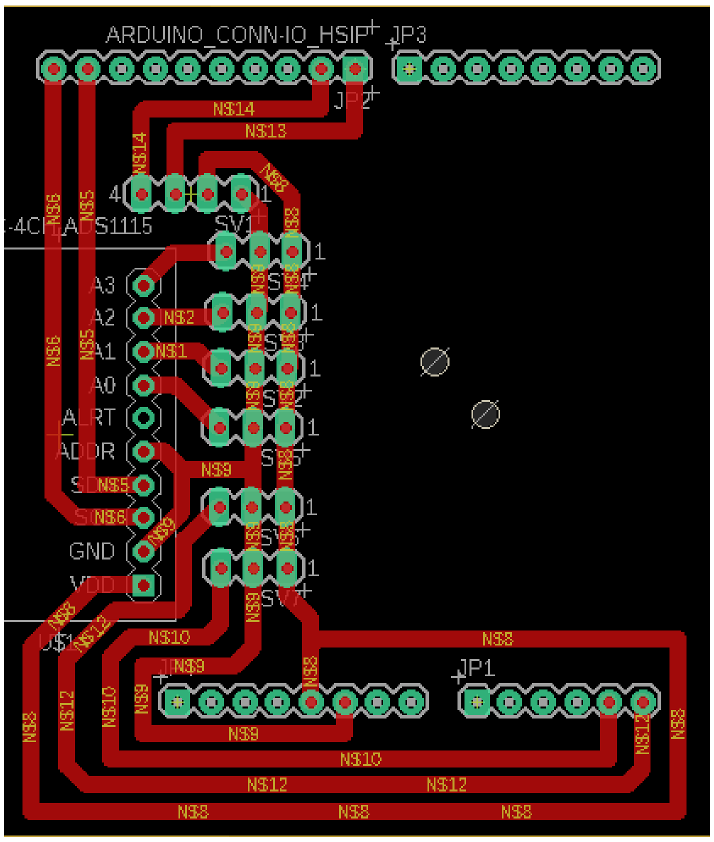


Figure B.1: Final version of the PCB.

Appendix C

States Visited

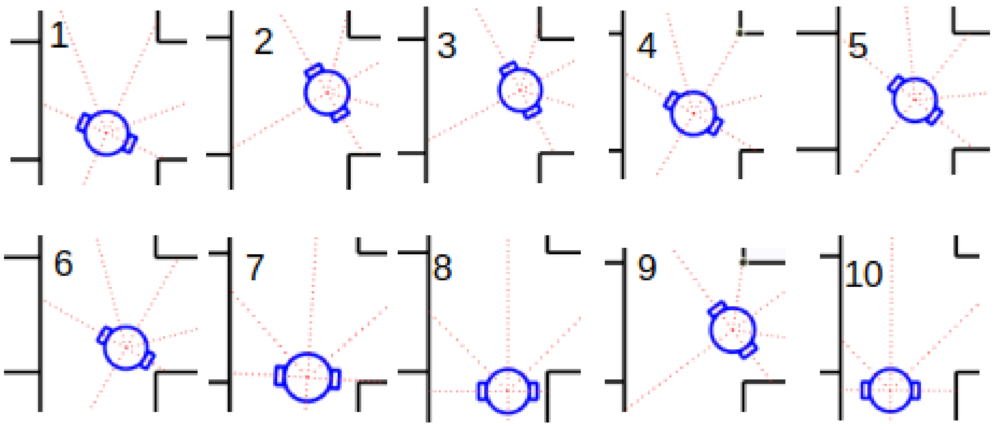


Figure C.1: States visited - Right turn on a door - Map 1.

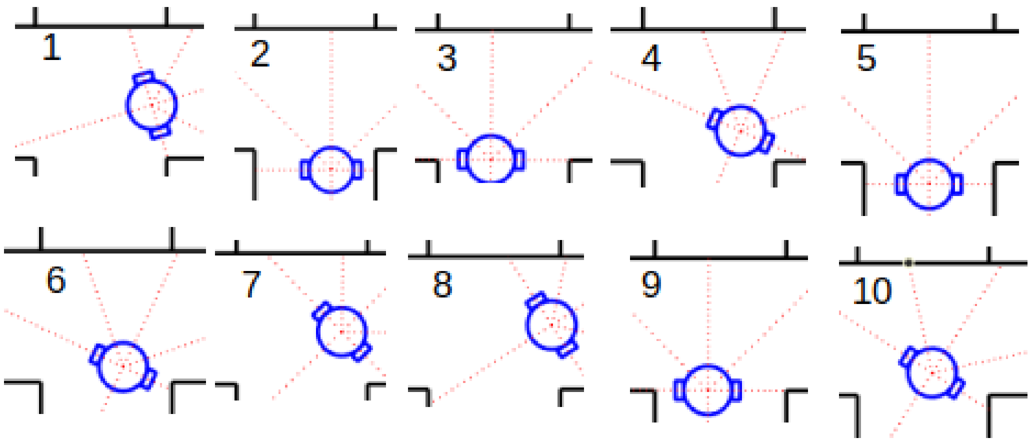


Figure C.2: States visited - Right turn on a door - Map 2.

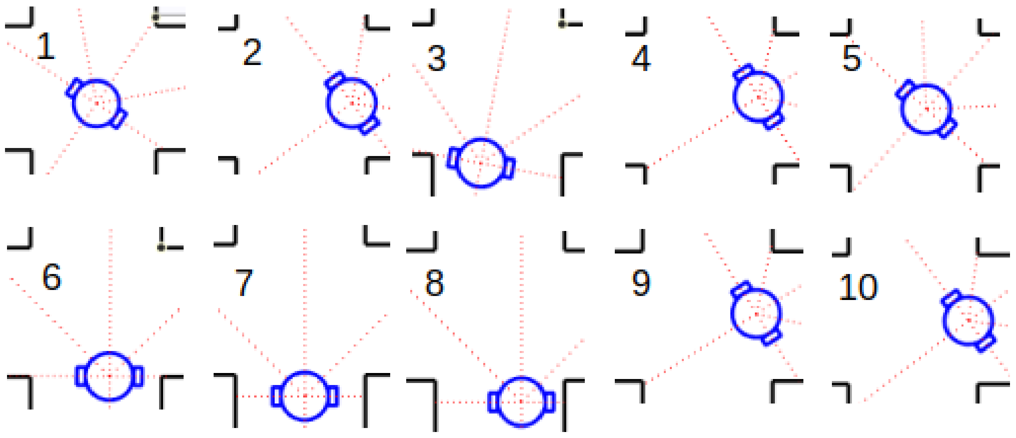


Figure C.3: States visited - Right turn on a door - Map 3.

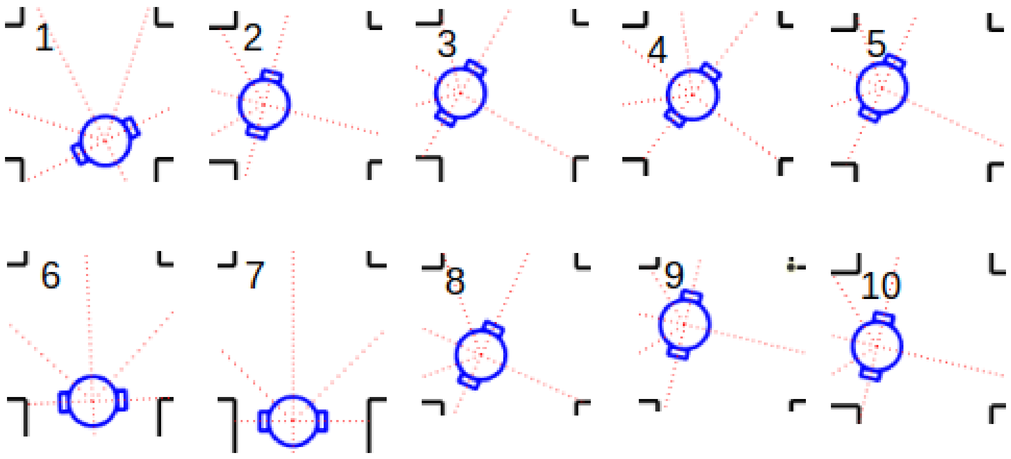


Figure C.4: States visited - Left turn on a door - Map 1.

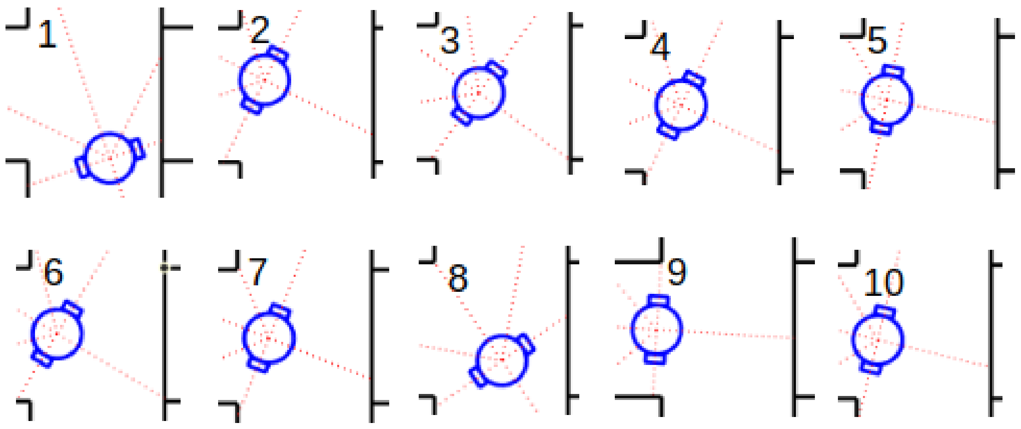


Figure C.5: States visited - Left turn on a door - Map 2.

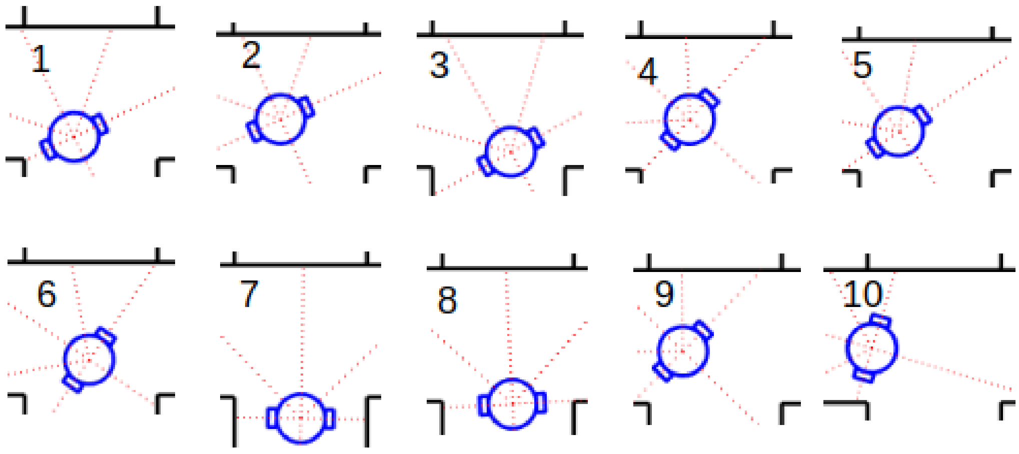


Figure C.6: States visited - Left turn on a door - Map 3.

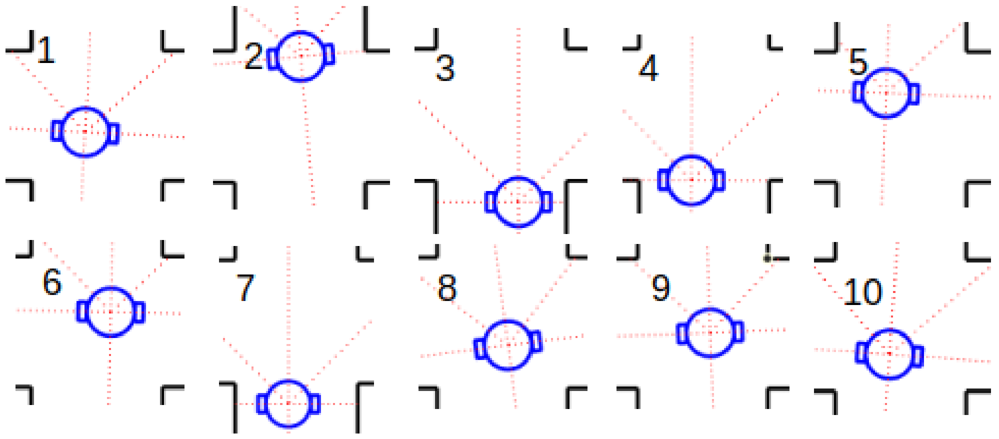


Figure C.7: States visited - Going Forward on a door - Map 1.

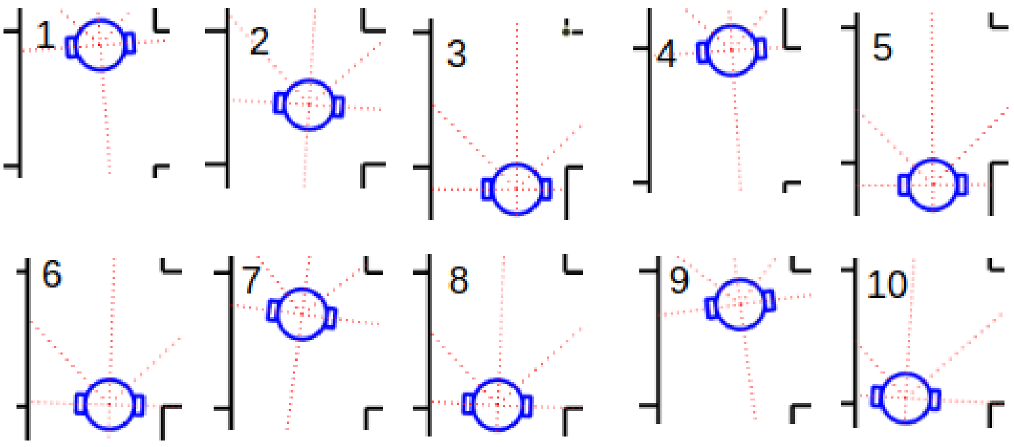


Figure C.8: States visited - Going Forward on a door - Map 2.

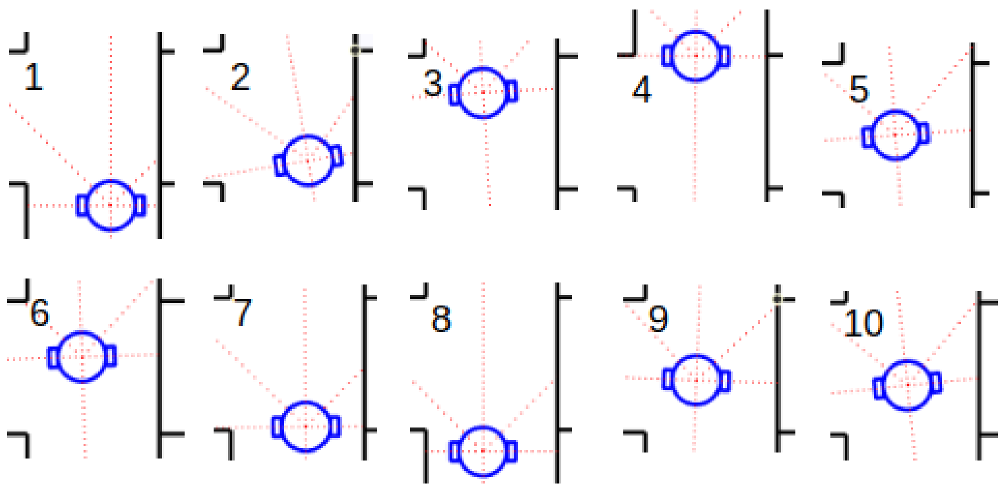


Figure C.9: States visited - Going Forward on a door - Map 3.