ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

Desarrollo de Capacidades de Aprendizaje por Refuerzo en un Agente Autónomo en un Videojuego

Development of Reinforcement Learning Capacities in an Autonomous Agent in a Videogame setting

Author/Realizado por
Pedro Jesús Reyes Santiago

University Lecturer/Tutorizado por
Prof. Dr. Marlon Nuñez Paz

Department/Departamento
Departamento de Lenguajes y Ciencias de la Computación

MALAGA UNIVERSITY, SPAIN (December 2016)

Resumen *Desarrollo de algoritmos de aprendizaje por refuerzo y su aplicación sobre un modelo desarrollado. El modelo desarrollado es un videojuego en el cual un agente aprende a jugar utilizando aprendizaje por refuerzo. La hipótesis inicial es que no supondrá un gran reto para estos algoritmos el modelo establecido ya que es un entorno pseudo-determinista. En cualquier caso el objetivo principal es la comprensión de los algoritmos clásicos de aprendizaje por refuerzo sobre un problema el cual podamos abordar de manera sencilla y conocer los fundamentos de esta rama de la Inteligencia Artificial.*

Keywords: reinforcement learning, agent

Abstract. *Development of reinforcement learning algorithms and their application to a model. The developed model is a game where the agent has to play using these algorithms. The initial hypothesis is that the model will not be too big a challenge for the algorithms due to the pseudo-deterministic features of the model. In any case, the end goal of this project is to attain the understanding of classical reinforcement learning algorithms in a problem we can deal with easily and absorb the basics of this branch of the AI.*

Keywords: reinforcement learning, agent

Table of Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Reinforcement Learning (RL) does not fit into any other learning algorithm as far as research has shown according to [1] where the authors introduce these algorithms as another type of learning, which cannot be included inside the set of supervised or unsupervised learning algorithms. Nevertheless, there does exist a relation between RL and Dynamic Programming (DP) as well as with Monte Carlo Methods. Indeed, RL is a middle point among these two computational models. RL is the combination of these two computational models. RL algorithms can use a model to fit better their predictions however this model is not mandatory for making them work. In fact, as shown in many cases, the usage of a model may lead to major errors in the estimates due to dynamic environments, for instance, where the action policy to be carried out may change drastically over the time period. The use of models in RL algorithms is known as the planning phase. This phase consists of focusing a fraction of the learning on making the RL algorithm interact with the implemented model, in other words, so when it faces the real model the estimates will be more accurate.

RL algorithms in their classic form, which will be what we will use in this project, consist of a table that indicates in its x-axis the possible states of the environment and its y-axis the possible actions to be taken. Each pair state-action is associated with a value or estimate so-called Q-value. Each Q-value indicates the estimation about the future reward that will be obtained in case a particular action is taken in a particular state. From experience gained through the taking of actions in many different states, this will enable the estimates to be more accurate.

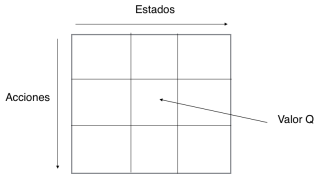


Fig. 1: Table of Q-values.

In Figure 1 we see the Q-table used in a classic reinforcement learning algorithm where states are indicated by the x-axis and the actions by the y-axis. Each cell indicates the Q-value associated with that pair state-action

The environments described by a sole agent in RL algorithms are defined as Markov Decision Process (MDP). A finite MDP is a record $< X, U, f, \rho >$ where X is the set of states in the environment, U is the finite set of actions to be taken by the agent, $f : X \times U \times X \to [0, 1]$ is the transition probability function among states, and $\rho : X \times U \times X \to \mathbb{R}$ is the reward function.

$$R_x = E\left\{\sum_{j=0}^{\infty} \gamma^j r_{k+j+1}\right\} \tag{1}$$

The goal of RL is to maximise the long-term reward. This cumulative reward maximisation comes through the interaction with the environment. Equation 1 describes the calculus of the

future reward estimation. Such estimation comes from the reward obtained in time j applying over it a discount factor γ. The discount factor γ specifies the importance of the expected future rewards, in this way if we observe the Equation 2, we can easily perceive the importance of this parameter. For values from 0 to 1 we would have that for a value of zero, the estimation would not consider the current reward, whereas a value of 1 would give a tremendous importance to long-term rewards.

$$Q_{k+1}(x_k, u_k) = Q_k(x_k, u_k) + \alpha_k[r_{k+1} + \gamma_k \max_{u'} Q_k(x_{k+1}, u') - Q_k(x_k, u_k)] \tag{2}$$

Equation 2 makes reference to the estimation updating of the Q-values in the RL algorithm so-called Q-Learning. In this equation the following parameters co-exist:

- $Q_{k+1}(x_k, u_k)$: new Q-value for a state x_k and an action u_k. Based on the value for each pair state-action, the priority of one action over another is set in a particular state. Actions with a bigger Q-value will be more prone to be selected.
- $Q_k(x_k, u_k)$: old Q-value for a state x_k and an action u_k.
- α_k: learning rate used in time k. This parameter makes reference to what extent the new information, that is, the current estimation and the reward obtained, replaces the old estimation for a pair state-action. In environments that are completely deterministic a learning rate of 1 would be the optimal value since the new value, not affected by stochastic aspects, would replace the old estimation completely. If we were in a stochastic environment, then the optimal values for α would be close to 0. This parameter may be modified temporally decreasing or increasing its value with respect to the period of time.
- r_{k+1}: is the reward reaped for executing action u in state x_k.
- γ: discount factor used in time k. This parameter determines the importance of future rewards. Values closer to zero place importance on immediate rewards, whereas values close to one do it over long-term rewards.
- $\max_{u'} Q_k(x_{k+1}, u')$: maximum Q-value for the next state x_{k+1} and any possible action u'.
- $r_{k+1}+\gamma_k \max_{u'} Q_k(x_{k+1}, u')$: this addition sets the learned information as fruit of the interaction with the environment.

Equation 2 will be used in this project to give learning capacities to our agent. Q-Learning is considered a model-free algorithm due to its freedom with respect to the reward function and the transition probability function as we can see in Equation 2.

The goal of this dissertation is to acquire knowledge in the field of RL throughout the development of an agent in a game. The main process of the whole project will be the development of these RL algorithms and the analysis of the different aspects that encompass them. Thereby, our main goal is to achieve the greatest performance that may be possible through the experimentation and analysis of the results obtained by the codification and adaptation of these algorithms to our game.

The model that our agent will have to learn is a game of warlike features. The game consists of a map, initially of size 3×3, where each cell contains particular features. In some cells, the agent is attacked, whereas in other cells the agent is attacked and can also attack back. In other cells, the agent can pass by without being attacked while in others, the agent can recover its life for returning back to fight later on. Our agent will have to discern between which cells are better in each moment to accomplish the final goal; which is to destroy the enemy.

The origins of this dissertation stem from a personal motivation for the acquisition of knowledge about RL algorithms. This mission approach, I view to be an investment in order to deepen my

knowledge of RL algorithms for the further development of future computational models in this field or its multi-faceted application in any other field.

The framework of this document is Section 2 titled Background where the history of RL will be illustrated. Section 3 titled Key concepts where key concepts of RL will be introduced for helping the reader. Section 4 titled System where we explain the model where the RL algorithms have been applied. Section 5 titled Solution where we explain the different algorithms we have used in the model and the results following. Section 6 titled Global Analysis where an analysis of the results obtained with the best results is presented. Section 7 titled Conclusions and future work where we draw different conclusions from our experiments and the knowledge acquired from the same.

2. Background

Nowadays RL is a studied and wide field as we can see in [1] and [2]. RL attempts to imitate the way humans and animals learn new skills and concepts such as playing chess, basketball or running. A general problem in many of this skills is the presence of many states, where each state contains a subset of characteristics of the environment. For instance, in basketball, shortening the problem to its simple form, we would have as many states as forces we could apply to the ball. The more states we have, the more precise we will be but the longer the time we will need for learning the optimal behaviour. Emerge at this point the well-known problem in AI so-called curse of dimensionality.

In this research field, there are two main problems in its basic form. The first problem is the curse of dimensionality as the more state we have, the more time the algorithm will need for adapting the estimates to a particular problem. The second problem is implicit in the way RL works, that is, the slowness produces by the non-label interaction with the environment. The interaction with the environment is made by rewards given to the agent, so there is nothing such as in artificial neural networks where an input is classified as good or bad. Therefore, we have two main problems in RL algorithms: the curse of dimensionality and learning speed.

The curse of dimensionality occurs in problems where the states may be split in a continuous manner such as the position of a lever, which must be lifted up in the vertical axis, being tied to a car that has to go to the left or to the right to keep the lever in the vertical axis. The learning speed of a RL tends to go down to the extent that the number of states goes up. This problem may be resolved with the use of other supervised learning algorithms such as artificial neural networks (ANN). A clear example of it is [3], [4], and [5]. These two last use several layers of neurons where each layer supply part of the information given at the inputs.

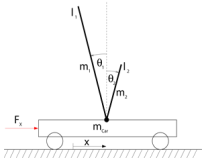


Fig. 2: Cart pole balancing.

In Figure 2 we see the cart-pole balancing problem, a common problem in RL where a cart must keep in the y-axis a pole which is tied to it accelerating to the right or to the left. This problem illustrates the curse of dimensionality.

RL has a characteristic that may be viewed as a problem and as an advantage. The non-labeled information makes the algorithm being able to learn with less information than supervised learning algorithm while at the same time makes harder the learning. For this issue, it exists a wide variety of methods with the purpose of diminishing this time. Obviously, depending on the problem, some methods will be better than others:

- Lambda temporal-difference methods [6].
- Actions selection methods such as ϵ-Greedy, soft-max or UCB (Upper Confident Bound).
- Descent-gradient methods [7].
- Learning methods such as Q-Learning or SARSA [8].
- Heuristic search in the space of states making use of methods like Monte Carlo Tree Search [9].
- Learning by imitation [10], which consist of a boosting at the beginning of the algorithm execution for helping the agent to know better its environment from the beginning and carrying out a better performance in long-term, even if this environment is dynamic as their authors claim.

The different elements that we have to our reach in this research field are a lot, and in many cases, a few of them may work nicely for the problem. In other cases, the complexity of giving a solution to the problem relies more on the representation than in the use of one method or another. In this last case, the search of the best representations is, in many cases, more of an art than a science.

RL algorithms had its start in industrial control systems and video games. Such start it is set by the program developed by Arthur Samuel for playing checkers in 1952 [11] followed by an improvement in 1967 [12]. This program will be later on executed in the scientific computer IBM 701. Although Samuel did not define RL formally, he did establish some features of it about the idea of differential learning methods which is the core of RL algorithms. [3] It was in 1992 when these algorithms reached a new height due to the program developed by Tesauro, so-called TD-Backgammon, in which a mix of ANN and RL was carried out. This algorithm was able to beat professional players and it is worth noting that there exist more professional players in backgammon than in chess. Thereby, it is more remarkable the fact that this algorithm was able to accomplish such a huge step forward. TD-Backgammon combined the use of an ANN with 198 input neurons, representing each neuron a straight connection with each position of the board, and 2 output neurons pointing out the probability of victory for each player. The weights among neurons were modified by a $TD(\lambda)$ algorithm. The $TD(\lambda)$ algorithms are RL algorithm such as Q-Learning which include a parameter λ that stores how many past estimations product of the interaction with the environment must be taken into account for the calculation of the current estimation; this is known as eligibility traces.

RL has not just being applied to video games but to other fields like mechanical engineering where we found tasks such the acrobot [13]. In this task, a robot made up by the union of two lines with a torque point in the joint among them must each a specific height with the tip of its feet applying a positive, negative or null torsion over the torque point. The representation Figure 3 has been extracted from [1]. RL has proved to be useful in others spheres such as the planning movement of elevators [14] or the usage of available bandwidth for offering a better mobile service [15].

In Figure 3 we see a typical problem in mechanical engineering that manifests the curse of dimensionality. This issue shows up due to its multiple states that can be found both in the angle between the two lines and the actions applied alike, that is, apply a positive, negative or null torque over the torsion axis. Besides, this problem shows how RL can be applied to many different fields.

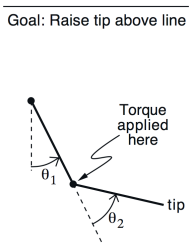


Fig. 3: Acrobot.

[16] Multiagent systems are another field where the framework expressed by RL algorithms is playing an important role under the name of Multiagent Reinforcement Learning (MARL) systems. In this field has already being researched key factors such the stability and adaptation of agents to different levels as for instance the learning convergence when others agents remain stationary [17] [18]. Many of the concepts discussed in MARL systems are later on showed in a practical way in events such as the RoboCup [19], an annual event where specialised robots carry out different tasks in different environments for beating records or other robots [20].

Currently, exist a new type of algorithms known by deep learning. These algorithms make use of ANN with several hidden layers of neurons among the input and the output layer. The overall idea consists of applying a stochastic gradient descent on the learned information and based on these gradients modify later on the weights of the neural network. This concept consists of identifying a loss function, which establishes the difference between the forecast and the real output. Based on this loss function an error is back-propagated through the whole neural network obtaining during the process an increase or decrease of the weights in the synapses that exist in the connections between neurons. This algorithm combined with other techniques such as search trees or RL algorithms have given good results on a wide variety of problems such as different Atari games where the same computational model is able to reach an optimal behaviour without changing any parameter value [21]. [22] The last hit was in January 2016 where for the very first time a machine was able to defeat different master players in the game Go, which is impossible to resolve by brute force due to its huge branch factor.

[22] Makes use of a heuristic search algorithm so-called Monte Carlo Tree Search (MCTS) which has been used ever more since 2006 in every type of board games such as Go or even in real time games such as Rome Total War II [23]. The core ideas of MCTS were presented by Coulum [24] and Kocsis and Szepesvári [25]. MCTS is specially suitable for those problems where we can develop a model of the environment and which model calculation is low. This heuristic is based on reaching a final node as fast as possible assessing the path through different explorations realised. Such assessment is established by weighting each node and focusing the learning on those nodes that must be more explored.

Finally, there is a narrow relationship between RL and psychology. Many theories applied over RL algorithm emerge from theories of psychology applied in animal learning experiments. In RL exist mainly two categories that are: forecasting algorithms and control algorithms. In the first one, the values are estimated and in the second one there exist a relation based on a value between states and actions. These two categories have their equivalent in psychology known as classic and instrumental conditioning experiments, respectively.

In psychology, in a classic conditioning experiment, an animal is exposed to a stimulus that is thrown each time an action is carried out by the animal. In an instrumental experiment, the situation changes as the animal is fed with something that it likes or dislikes depending on what it does. From this last type of experiments, it is drawn that the animal tends to carry out those behaviours that are positive rewarded while, by the opposite, tends not to perform those behaviours that are negative rewarded. [26] The Effect Law of Throndike is inspired by these experiments. This law causes a split in the psychology field between those researchers that support that actions are carried out by animals based on random actions [27] while others consider the existence of previous knowledge or a certain level of reasoning. Among the researchers with greater influence on this theory we find Cark Hull [28] and B.F. Skinner [29] whose take as a start point of their research the selection of behaviour based solely on the consequences of the actions.

[30] The concept of model-free RL algorithms is present in psychology too. The disparity between these two in psychology is known as control of the behaviour lead by targets or habituation. This habituation is something that emerges from the stimulation while behaviours led by targets emerge from an understanding of the actions values. These values set a direct relationship between executed actions and their consequences. [31] sets that animals do not use either one model or the other but both at the same time. Both models offer an action, and one of them is selected based on the trust on that action.

[32] The theory of temporal differential methods is also present in psychology where Rescola and Wagner carried out in 1972 a model so-called temporal-differential model of the classic conditioning. This model sets a relationship between conditioned answers and non-conditioned answers as well as their response time, including also the stimulus times, and emerging so one of the models more widely known in the theory of classic conditioning in psychology.

3. Key concepts

Reinforcement Learning is a framework of general purpose for the development of IA. It is designed for making an agent execute different actions in a world. These actions lead the agent to discover new states in the aforementioned world at the same time that the agent obtains a reward based on the state and action taken. The goal is to maximise this reward. Therefore, the basic key elements of the development of these algorithms are: states, actions, and the rewards obtained when executing an action in a particular state.

Approaches there exist three possible approaches to this model: policy-based (we look for the greatest future reward given a policy); value-based (we look for optimal value function for any given policy); and model-based (we learn some knowledge from a model which implements the transitions of a real model).

State-action value function $Q^\pi(s, a)$ returns the total expected reward being in state s and executing action a when using π as action policy, that is, $Q^\pi(s, a) = \mathbf{E}[r_{t+1}+\gamma r_{t+2}+\gamma^2 r_{t+3}+\ldots \mid s, a]$.

Discount factor (γ) measures to what extent the search of a reward is made. In other words, the discount factor set the importance of future rewards over the immediate reward. In this way, a value close to 1 will set the search of a long-term reward whereas a value of 0 will place the focus on the immediate reward. The value of the discount factor γ is set in the range between 0 and 1.

Exploratory factor ϵ is the parameter in charge of to what extent part of the actions are taken randomly. This parameter is vital for the good performance of the algorithm as it can set the

difference between a fast learning, a progressive learning or the absence of such learning. The value of the exploratory factor ϵ is set in the range between 0 and 1.

Monte Carlo methods their estimations are made at the end of the episode, that is, when the environment is in a terminal state. It does not assume any knowledge of the environment and the estimations are carried out based on the experience acquired by the interaction with the environment. Such estimates are based on the average of rewards obtained.

On-policy and off-policy there exist two different temporal difference methods. The off-policy methods which look for an optimal value function for any action policy, whereas the on-policy methods look for an optimal value function for a given action policy.

Policy π is a behavior function, which establishes that for a state s an action a must be executed, that is, $a = \pi(s)$.

Markov Decision Process (MDP) is a set $< X, U, f, \rho >$ where X is a finite set of states of the environment, U is a finite set of actions to be carried out by the agent, $f : X \times U \times X \to [0, 1]$ is the transition probability function, and $\rho : X \times U \times X \to \mathbb{R}$ is the reward function.

Dynamic Programming (DP) based its estimations in a perfect model of the environment which fulfil MDP constraints.

Q-Learning is an off-policy temporal difference method which state-value function is $Q(s, a) = Q(s, a) + \alpha[r + \gamma * \max_{u'} Q(s', u') - Q(s, a)$.

Learning Rate α sets to what extent the information received when carried out an action in a model overwrites the current information. Values of α are between 0 and 1. A value of 0 would make that the information received not being overwritten at all and thereby there would not be any learning. A value close to 1 means that the information overwrites the current information completely. The speed of learning will change based on the stochastic factors of the environment.

SARSA State-Action-Reward-State-Action (SARSA), is an on-policy temporal difference method which state-action function is $Q(s, a) = Q(s, a) + \alpha[r + \gamma * Q(s', a') - Q(s, a)$.

Task, episode, step a step is an action carried out by the agent in a particular episode. An episode is a set of steps, which are given in a particular order and are finite. A task is a set of episodes. An episode may be a game where each step is an action carried out by each player in that game. The execution of 5 games is a task made up of 5 episodes. In this project, we generally will use executions of 100 tasks with 100 episodes each task.

Episodic and continuous tasks An episodic task defines a start and an end whereas continuous tasks are infinite or the end is unknown. In an episodic task the reward for the executed actions could be given at the end, that is, when we already know what happened (e.g. a positive or negative reward could be given in a game when we know if we lose or won, respectively) whereas in a continuous task the rewards must be given throughout the time period since in this kind of tasks we do not know the end of the task. In this project, we will carry out an episodic task.

Temporal difference learning is a mix of the concept of Monte Carlo methods and Dynamic Programming. Some temporal difference methods are Q-Learning and SARSA.

4. System

In this section we will present the technologies used for the development of the system (4.1. Technologies); a detailed description of the model that the agent will have to face (4.2. Model); the interface developed for the visualization of a game in which the agent will carry out actions to win a game against the aforementioned model (4.3. Interface: local interface and web page); and we will end this section with the UML diagrams of mandatory nature in any software project (4.4. System diagrams).

4.1. Technologies

We have used the lambda functions of Java 1.8 [33]. The developed algorithms, as well as the model, have been coded in Java.

Additionally, pouncing on the concepts acquired in Aspect-Oriented Programming (AOP) during the master we have made used of the AspectJ framework. The official web page of this framework is [34] and in [35] can be viewed a brief description of the aforementioned framework. This framework offers an AOP of the Java language. The AOP of AspectJ consists of the execution of code chunks that are executed when a particular call to a method o even when instancing an object of a particular class is made. With this paradigm, we can make aspects of our code that are not particularly specific to a class but a functionality of general purpose based on the separation of file aspects .aj. These files would execute code that represents a general functionality of the application making more legible the code since this one it is not intertwined with other aspects of the system in the same file.

An easy example to see is the security of a system. Each time we log in a web page we must check that we are logged in and the system must be checked this too for each button on it. Without AOP we would have to go over class by class and set a code chunk for checking that the user is logged in. With AOP we would have to declare an aspect file that arranges all the methods that must check if a user is logged in and with a condition of type "around" we would execute such piece of code without going method by method setting this piece of code and reinforcing the legibility of the code, the time of development, and the ease of testing . Therefore, for this project we have used AspectJ for tasks such as:

- Capture of exceptions which are very often.
- Debugging. In some cases for testing, we did not want to make the output of some classes appear in the console. Because of this, we hide with an aspect the output in the console of classes that we are not interested in debugging.
- Results backup. With AOP we have been able to easily save all the results of our algorithms without intertwined backup-related code with reinforcement-learning-algorithm code. With just one aspect we save the results of our algorithms having at the same time a clear reinforcement learning classes with no code about the backing up of the results. This allowed us a bigger legibility of the code and being able to focus independently on the one side in the code of the developed algorithm and on the other side in the backup of the results produced by these algorithms.

For displaying the results, we have made used of the statistical programming language R [36]. With this programming language, we have plotted the vast amount of raw data produced by the developed algorithms. We have connected Java to R making use of a server so-called Rserve [37] created on the local machine for automating the process of creating results in plots. Rserve is a TCP/IP server which eases the connection between R and other languages. With this connection, we can make calls to the programming language R in such a way that once the Java code of our algorithms is done, the R code is called and execute over the raw data files the creation of plots. Therefore, there is no need to use an IDE such as RStudio for manually creating these plots once the algorithms have finished.

Making use of what we have learned with the R language, we have expanded a bit more our knowledge of this language using the so-called platform Shiny. Shiny is a framework in which we can include our R code for creating plots in such a way that we can develop our web page and make

public our research. In this way, we make the dissemination of our results directly available online. With Shiny there is no need of previous knowledge about HTML, CSS or Javascript. Nevertheless, it is always recommendable to have a basic understanding of this languages.

With this page, we want to make clear one of the concepts we consider more important learning during the master: experiments should be replicated whenever someone wants. Thereby, we have created this web page `https://research-results-pedro-reyes.shinyapps.io/ResearchResults/` where it exists a copy online of this document, a section for downloading the code, instructions for executing the code, including the model we have created for the different developed algorithms and, additionally, once you can execute your experiments you will be able to display your results through the web page based on the raw data files generated by your local machine. With all this code available online you can do your experiments and corroborate that the results included in this document match with the reality.

Finally, for the interface, we have used Processing [38]. Processing is a programming language based on Java and that it can be used in Java, Arduino, Javascript, and so on. This language is oriented to graphic art allowing to code easily graphic interfaces setting a drawing framework where the user has total freedom for drawing.

4.2. Model

In RL there exist the so-called n-armed bandit problems. In these problems, there is a unique state that has associated several actions. In this problem, there exist n-armed bandits and a player or agent that has to choose which slot machine should play more, that is, there as much actions as slot machine we could play. In many cases, these problems use a normal distribution for the rewards given to the agent for each slot machine, so the agent will have to choose the slot machine that on average give the highest reward. In our model, in this case, a game, we look for the same thing than in an n-armed bandit problem with the disparity of having more than one state. We have exactly the same situation than in the n-armed bandit problem but we do not only learn the best slot machine of one casino but all of the casinos in the city.

The model developed is based on a map of $n \times m$ which independently of the size will follow the same logic. This logic is the following:

- Top-right corner is where the enemy is.
- Top-left corner is the bomb shelter.
- Down-left corner is the repair zone.
- The border of the map, except the top-right corner, are zones where the agent is not attacked. To the contrary, the central zone is where the agent should not ever be since this an ambush zone where the agent will be attacked, and it will not be able to attack back.

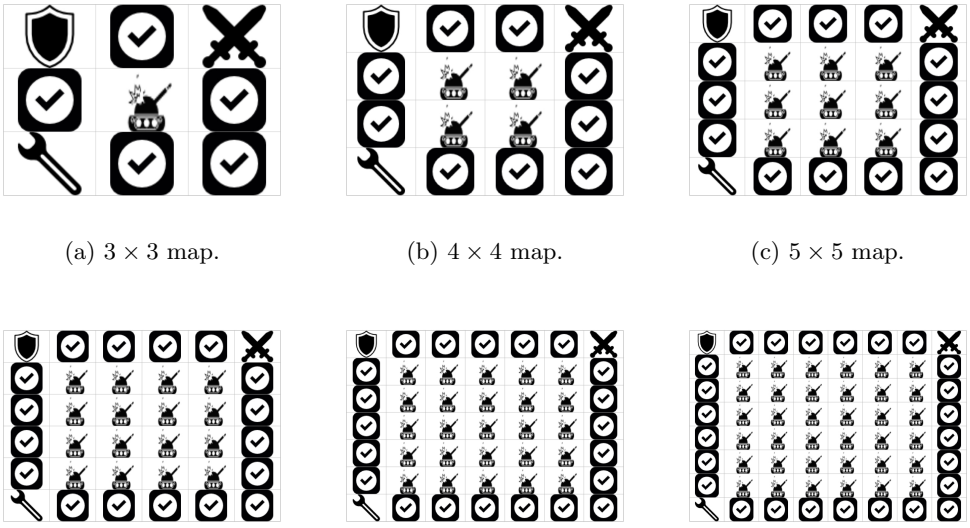
Therefore, the ideal logic of the agent should be to go to the borders of the map and reach the top-right corner where the enemy is. In the case of having little life and not being able to kill the enemy, they agent should come back through the borders of the map and get its energies back in the repair zone. Finally, in case that an air attack occurred the agent should shelter in the bomb shelter.

Each cell of the map has associated a state which can be found in Table 1. The state may describe the type of zone or the state of the agent.

In Table 1 we can see which would be the zones that our agent would find for the different sizes of the map in Figure 4. Figure 4 shows how the same distribution is always applied to the maps.

	State	Description
	Safe zone	Describes a zone where the agent does not attack and it is not attacked neither.
	Repair zone	Life of the agent goes to 100%.
	Ambush zone	The agent is attacked but can't attack.
	Attack zone	When the enemy is in this zone the enemy attacks it and the agent attack the enemy back. The goal is to get the enemy's life to zero.
	Bomb shelter	It is the zone where the agent in case of dying because an air attack could be survive. In this zone the air attack does not hurt the agent
	Receiving an air attack	The agent is receiving damage in form of air attack. When this happens the agent is attack in every single zone of the map except in the bomb shelter.
	High life	The agent has more than 50% of its total life.
	Low life	The agent has less than 50% of its total life.
GAME OVER	Agent destroyed	The agent has been destroyed by the enemy, that is, its life is equals 0.
	Enemy destroyed	The enemy has been destroyed by the agent, that is, its life is equals 0.

Table 1: States of the game.



(a) 3×3 map. (b) 4×4 map. (c) 5×5 map.

(d) 6×6 map. (e) 7×7 map. (f) 8×8 map.

Fig. 4: Different sizes of the map for the same game.

A central part where the agent is attacked but can't attack, a top-left cell that is a bomb shelter, a top-right cell that is an attack zone, and a down-left cell that is a repair zone.

To the extent we increase the size of the map the complexity for learning the same games goes up. In a 3×3 map there are 58 states, 100 in a 4×4, 155 in a 5×5, 218 in a 6×6, 295 in a 7×7 y 385 in a 8×8. Each state has associated 8 actions (go to the right, left, top, bottom, top-right, top-left, bottom-right, bottom-left). Some of the actions may not be possible to execute depending on the position of the agent in the map. For instance, the agent will not be able to go to the right if it is in the top-right cell of the map.

The damage that is done in each zone depends on the size of the map. Only the attack zone does not depend on this magnitude since the damage that the agent or the enemy realise in the attack zone is always the 10% of the total life of the opposite. In particular, we have set that the agent and the enemy alike have 10 points of life so for each step that they are in the zone attack they received 1 point of damage. Concerning the damage realise in each step of the game in the ambush zone this is calculated by Equation 3 where $agentFullLife$ makes reference to the maximum life that the agent has got and $mapDimensionX$ to the size of the map in its x-axis.

$$damageAmbushZone = \frac{0.1 \times agentFullLife}{mapDimensionX - 2} \tag{3}$$

For the damage of an air attack, we use Equation 4 which makes the damage to scale according to the size of the map, that is, the same thing that with the ambush zone.

$$damageBombardeo = \frac{0.1 \times agentFullLife}{mapDimensionX} \tag{4}$$

With respect to the air attack, the number of steps that lasts an air attack we have established that will be equal to the size of the map and the start point of the air attack will be when the enemy has got 80% of its total life.

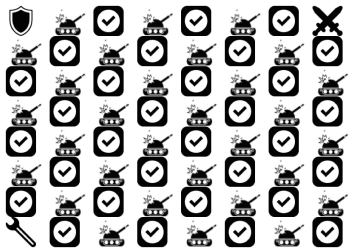


Fig. 5: Chess map.

As for the last benchmark of our results, we will set our agent on a map that we did not use for looking the best values of the parameters in the algorithms. This map will be named as chess map and will follow the structure that we can see in Figure 5, that is, we will keep the attack zone, the repair zone, and the shelter bomb in the same place whilst the ambush zones will be distributed along the map following the pattern of a chess board.

4.3. Interface: local interface and web page

Given the random factor that influence in these algorithms we didn't reach to any clear conclusion about whether a state-action policy is able to win a game or not so we have developed a minimalist interface to allow us to check if the AI acquired is enough to let the agent win a singular game.

Fig. 6: Gameplay of a 3×3 map.

This interface uses the same icons that we have seen in section 4.2 and the looking of it can be seen in Figure 6. The interface displays a map with the difference zones of the map, the position of the agent as a green tank (down-middle cell), the position of the enemy as a red tank (top-right cell), the life of the enemy as a vertical red bar (to the left) and the life of the agent a vertical green bar (to the right).

We can see through this interface in general features the behaviour of the agent after the learning phase.

Additionally, we have developed a web page to put at others disposal the repetition of the experiments executed in this project for anyone who wishes to replicate them. The web page is displayed in Figure 7 and it can be found at `https://research-results-pedro-reyes.shinyapps.io/ResearchResults/`. We can find in this web page the next tabs:

- Master Thesis: copy online of this document.
- Code: download the code, follow the instructions and execute your own experiments.
- Check your experiment's results: once you have executed your experiments, select the files from this tab and display your results online.
- Gameplay: play around with our version of the game online where you can set the difference learning methods, action selection methods and the values of the different parameters of the developed algorithms.

4.4. System diagrams

In this section, we attempt to give a clear idea about how the code has been created so in the case of modifying it any person can know where the new code or the modification should be done in the existent code. The code developed is split according to the following packages with a clear purpose for each of them:

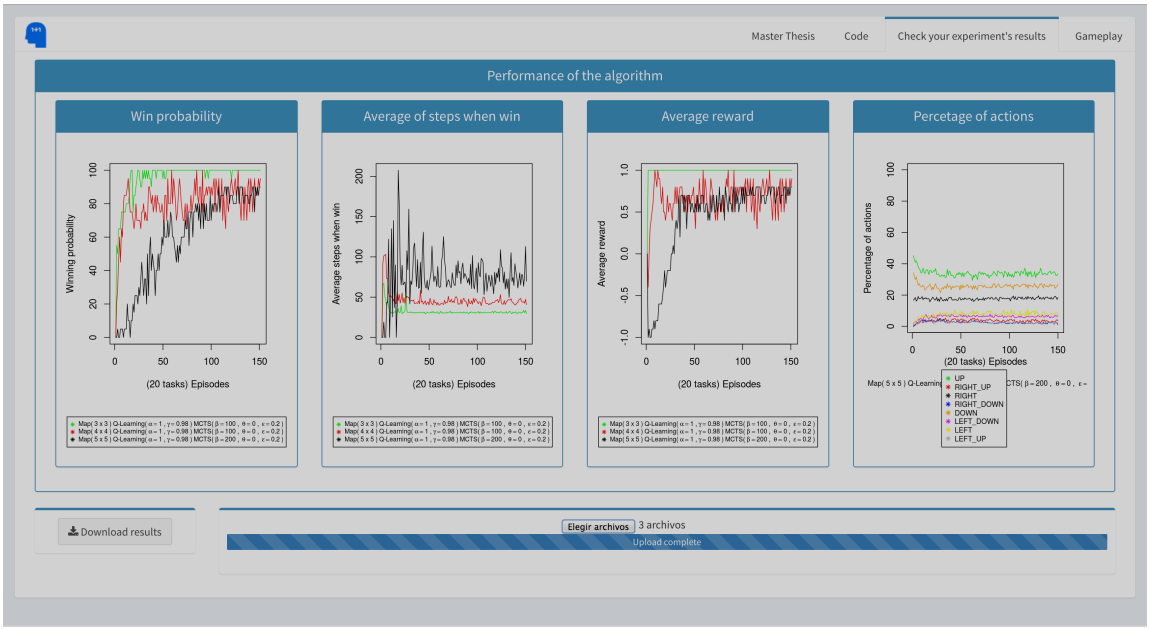


Fig. 7: Developed web page for experiment replication by others.

- Analysis: this package contains the Java classes needed for carrying out a communication between Java and R through the usage of RServe for the creation of plots. The R files are called from Java and they are placed in this package too. It is also provided in this package the main R file which is an example of what would be the R code that Java really executes for creating the plots.
- Aspects: contains the aspects created that were explained in Section 4.1.
- Auxiliar: this package contains some classes for helping other classes of other packages. Thereby, we can find the next classes:
  1. GameMap.java is for defining a map of the game and its size.
  2. Helper.java, which contains general functions such as obtaining, based on the parameters of an execution, the name of the file or setting the values of these parameters in the RL algorithms
  3. Parametrizacion.java stores the different values of the parameters that have been used in an execution.
- Output: this package contains solely a class so-called Output.java. The class set where should the data be stored, the name of the experiments, which kind of charts we can plot, which separator should be used between data and how many decimals should be saved in data.
- RL: this package contains the classes for executing the algorithms. RLearner.java contains the learning and action selection methods such as SARSA or Q-Learning. RLPolicy.java saves the state-action policy to be used by the agent. Action.java and State.java save the parameter needed for setting which is the action and state in the system. RLWorld.java is an interface, which is the one that must be implemented in case of wanting the agent to face another environment.

RLearnerConfigurationLoader.java is the class in charge of loading a previous learning in the system.

- WorldGameComplex: this package contains a class so-called GameWorldSimpleMap.java. This class sets how the environment works. The rest of classes in this package are for the execution of experiments, which follow a similar pattern all of them, so it is quite easy to do your own experiment just taking a look at their main code.
- YourTurn: this package is an empty world which class is EmptyModel.java. The methods must be implemented to execute your own experiments in your own world. You have got for executing your experiments a so-called class EmptyExperiment.java.
- GuiAndProcessing: contains the main classes for implement the interface, which are TankWorld.java and GUI.java. The first is used for GUI.java and the second prints the state of TankWorld.java into the interface.
- MCTS: the main class of this package is MonteCarloTreeSearch.java, which implements a Monte Carlo Tree Search. The rest of classes are used for the implementation and visualisation of the tree, which structure is implicit to the development of this algorithm.

The diagram of packages with the different classes can be seen in Figure 8

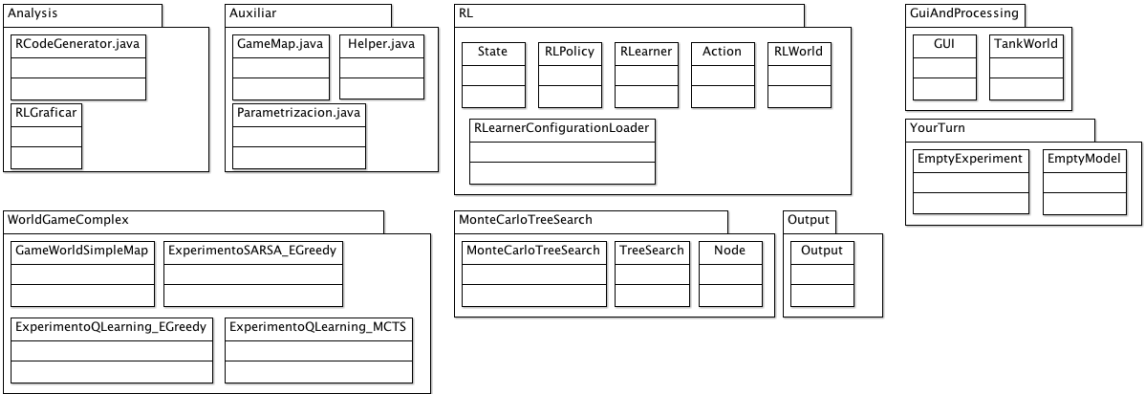


Fig. 8: Package diagram with the classes in them.

There would exist another project, which would be the developed web page, but due to that web page has only been developed for the purpose of making available the work of this TFM to other people and does not contain anything straight related with the goal of this project we have decided not to include it. Nevertheless, in the case of want to make use of this web page, you can go to it by the following URL `https://research-results-pedro-reyes.shinyapps.io/ResearchResults/`.

5. Solution

In RL there are 3 keys to obtain a good performance:

- Reward function: return a reward given a state and action taken by the agent.

- Learning method: establishes which is the way that the estimations are made.
- Action selection method: provide a way to select actions based on the knowledge learned.

We have established two reward functions for the algorithms. The first function is based on costs and benefits. This function will return a reward based on the zone that the agent is, the agent's life and the enemy's life with respect to its total life. This function can be seen in Table 2. We named this function as the continuous reward function. RAA means receiving an air attack whereas $!RAA$ means the opposite. The repair zone does not work if there is an air attack.

	State	**Reward function = benefit - cost**
	Safe zone	$beneficio = 0$ $cost = \frac{agentFullLife - agentLife}{agentFullLife}$
	Repair zone	$IF\ !RAA\ benefit = \frac{agentFullLife - agentLife}{agentFullLife},\ EOC\ beneficio = 0$ $IF\ RAA\ cost = \frac{agentFullLife - agentLife}{agentFullLife},\ EOC\ coste = 0$
	Ambush zone	$benefit = 0$ $cost = \frac{agentFullLife - agentLife}{agentFullLife}$
	Zona de ataque	$benefit = \frac{enemyFullLife - enemyLife}{enemyFullLife}$ $cost = \frac{agentFullLife - agentLife}{agentFullLife}$
	Shelter bomb	$benefit = 0$ $cost = 0$

Table 2: Continuous reward function.

The second reward function is discrete. This function does not offer continuous values as the first function. The values returned by this function may be defined as bad, good or without importance making a comparison with its possible values -1, 1 y 0, respectively. We named this function as discrete reward function. The function returns positive, negative or null values based on the zone where the agent is and taking into account also the life of the agent. The way this function works can be seen in Table 3. VA means high life, that is, the agent has more than 50% of life whilst $!VA$ means the opposite. In the case of receiving an air attack, the reward would be -1 except in the bomb shelter where it would be 1.

	State	**Reward function = R**
	Safe zone	$R = 0$
	Repair zone	$R = VA? - 1 : 1$
	Ambush zone	$R = -1$
	Attack zone	$R = VA?1 : -1$
	Shelter zone	$R = 0$

Table 3: Discrete reward function.

Next sections we analyse the algorithms Q-Learning, SARSA, and MCTS searching the best algorithm for our model using an empirical analysis taking as main measure the win likelihood.

5.1. Q-Learning

Description Q-Learning is one of the algorithms most used nowadays in RL. Q-Learning is an off-policy method. Off-policy methods can learn different behaviour policies. These algorithms are different from on-policy methods in which they are not based solely on the actions previously taken. In other words, off-policy methods may separate exploration from control whereas on-policy methods can't. This disparity gives as a result that off-policy methods can exhibit behaviours that were not necessary experienced in the learning phase.

Q-Learning follows Equation 2 to carry out their estimations. From this equation, we can easily pull out why this algorithm is off-policy. Even for action selection policies totally random, Q-Learning will keep trying of learning the optimal policy. Q-Learning needs two parameters for working, that is, the learning rate (α) and the discount factor (γ).

The learning rate and the discount factor are the parameters which we must be set for getting the best performance. In Listing 1 we can see how Q-Learning works using any action policy.

```
Initialize Q(s,a) // (Initially 0)
REPEAT(nEpisodes):
    Initialize s
  Choose a for s using Q
  REPEAT(until s is terminal):
      Choose a for s using Q
    Execute a, observe the new r and s'
    Q(s,a) = Q(s,a) + α[r + (γ * max_u' Q(s',u')) - Q(s,a)]
    s = s'
  END_REPEAT
END_REPEAT
```

Code 1: Q-Learning.

Each learning method, in this case, Q-Learning, have an action selection method. In our case, we have used firstly ϵ-greedy. This action selection method adds a soft way of selecting the action that is not solely selected based on the maximum value of the current policy. With soft, we mean that some of the actions are selected randomly following a probability of $1-\epsilon$. From a mathematical perspective of Equation 2 and assuming a continuous task we can make sure that at least mathematically speaking we will find the optimal actions for each state.

We have then for the experimentation of our algorithm the following three parameters for giving them values in search of the best performance: the learning rate α, the discount factor γ, and the exploratory factor ϵ.

Experimentation with continuous reward function The goal of the experiment is searching for the best values for the parameters so we can get the highest performance with Q-Learning using as action selection method ϵ-greedy and using as reward function the continuous reward function. The use of both methods makes a total of 3 parameters that are necessary to set up for searching the best performance. These parameters are the learning rate α, the discount factor γ, and the

Learning rate	**Discount factor**	**Exploratory factor**
1.0	0.01	0.01
	0.02	0.02
	0.1	0.1
	0.2	0.2
	0.4	
	0.5	
	0.6	
	0.7	
	0.8	
	0.9	
	1.0	

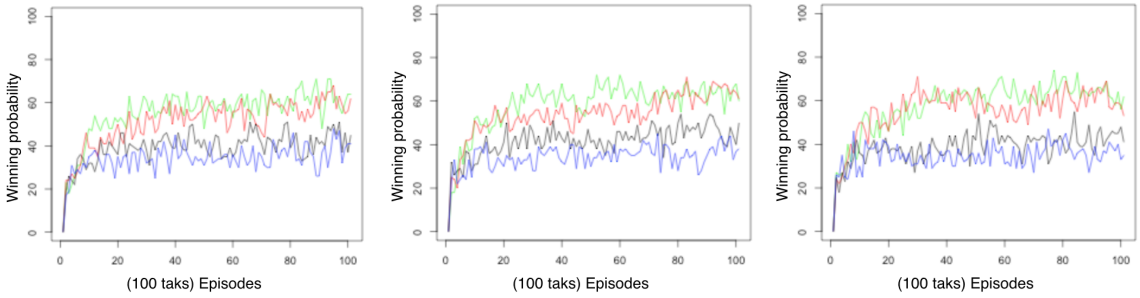
Table 4: Assessed values for a 3×3 map with Q-Learning.

exploratory factor ϵ. The values selected from the infinite set of values that we could have selected are established in Table 4.

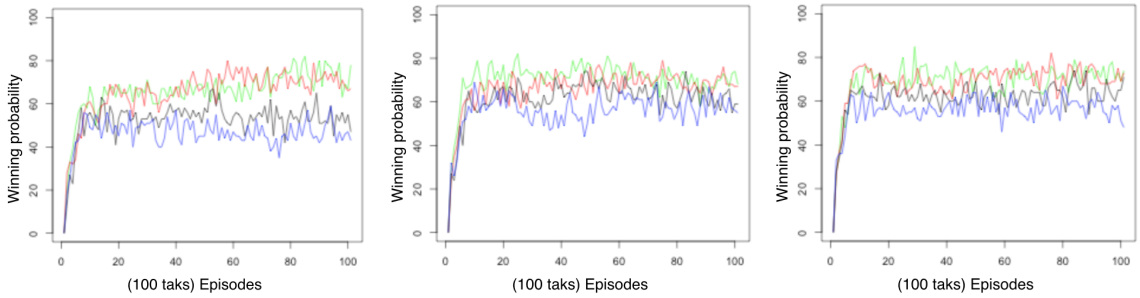
The chosen values pay attention to the studied bibliography and the logic. The model presented is a quasi-deterministic environment due to the air attack which time is not precisely described in the set of states. This imprecision in the state set is due to the sheer size of states that we could have if we describe with precision this set what would end up with the number of states increasing dramatically as the size of the map does. With respect to the rest of the information about the environment the agent is totally aware and each action u over a state s will lead the agent to the same state s' once and again. This quasi-deterministic environment makes that the best value for the learning rate is 1.0. Establishing this value to 1.0 we are saying that all the information obtained must not be undervalued as our world is not being influenced by stochastic factors. With respect to the exploratory factor, we have established that a high value of this parameter will lead to random behaviours so we set the range from 0.01 to 0.2 as possible values. Finally, the discount factor is the one that more uncertainty creates both logically and theoretically since we do not know the importance of the air attack that includes this stochastic factor we talked about before. It is because of this that the discount factor has a big range of values.

We will start with a 3×3 map for searching the best values of the parameters. In Figure 9 we can see the results of executing for the different values selected the performance of the algorithm based on the probability of winning a game. In the x-axis, we have the number of playouts whereas, in the y-axis, we have the probability of winning the n-playout. The probability of winning is calculated with the average of executing several times n episodes m times. In this case, we have executed 100 tasks and 100 episodes for each task, that is, 100 executions of 100 games each. This number of episodes and tasks have been chosen due to being a number high enough for displaying the convergence of the algorithm. The variable parameter in Figure 9 is the exploratory factor. The colour of the curve for an exploratory factor of 0.01 is green, 0.02 is red, 0.1 is black, and 0.2 is blue.

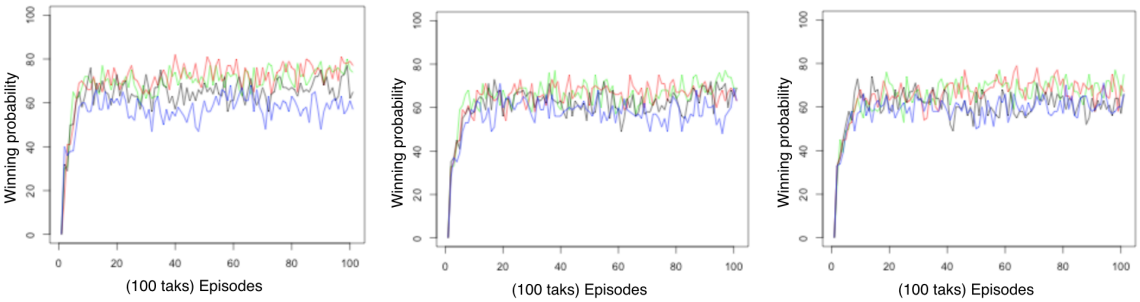
From the results in Figure 9 we can draw that for values below 0.4 in the discount factor exist a divergence with respect to the curve described by the different values of the exploratory factor whereas above this value, we can see a convergence of the performance independently of the exploratory factor. For values in the range from 0.4 to 0.9 we can see that, despite converging for



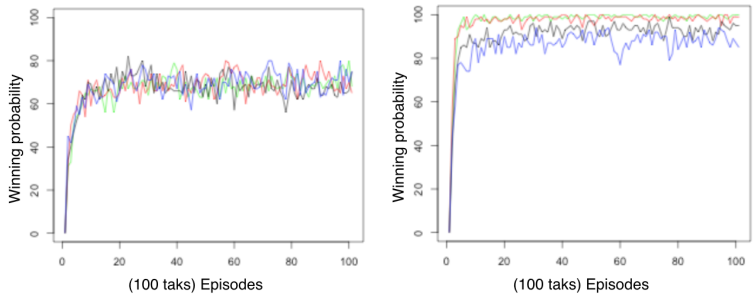
(a) $\alpha = 1.0$ $\gamma = 0.01$ (b) $\alpha = 1.0$ $\gamma = 0.02$ (c) $\alpha = 1.0$ $\gamma = 0.1$



(d) $\alpha = 1.0$ $\gamma = 0.2$ (e) $\alpha = 1.0$ $\gamma = 0.4$ (f) $\alpha = 1.0$ $\gamma = 0.5$



(g) $\alpha = 1.0$ $\gamma = 0.6$ (h) $\alpha = 1.0$ $\gamma = 0.7$ (i) $\alpha = 1.0$ $\gamma = 0.8$



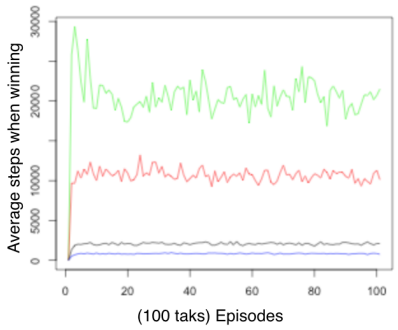
(j) $\alpha = 1.0$ $\gamma = 0.9$ (k) $\alpha = 1.0$ $\gamma = 1.0$

Fig. 9: Winning likelihood in a 3×3 normal map using Q-Learning and varying the value of ϵ.

the different values of the exploratory factor, the performance does not reach results greater than 70% staying around this performance during the whole aforementioned range. Finally, we can see that for a discount factor of 1.0 the probability of winning increase until almost reaching a 100%.

For a discount factor of 1.0 the performance is the best one achieved and one we have reached this point we only have one X left that refers to which is the best exploratory factor. As we can see in Figure 9 for the same learning rate and discount factor what we get is an overlapping between the values from 0.01 to 0.02 for the exploratory factor.

To reveal which of the values in the exploratory factor is better we will make use of another measure of the algorithm performance. This measure is the number of steps that the algorithm needed to win a game. In Figure 10 we can see the number of steps when winning a game for the values of the different values of the exploratory factor. We can see a different world from this view of the performance drawing that when we explore less, we need less time for winning. Nevertheless, this is wrong since we have to take into account that we are playing in a 3×3 map what affects to the probabilities of falling into the top-right cell, that is, the one in which we attack the enemy. These probabilities are bigger in this size of the map and because of this, the probabilities of winning the game having less exploration is bigger. We can see this in Figure 9 where the probability of winning a game for values 0.1 and 0.2 of the exploratory factor dwindles getting a probability of winning around 85%. This change in the exploratory value produces a disparity between the main measure established, the probability of winning, about a 15% for the smallest values like 0.01 and the biggest as 0.1. The colors used in Figure 10 are green for $\epsilon = 0.01$, red for $\epsilon = 0.02$. black for $\epsilon = 0.1$, and blue for $\epsilon = 0.2$.



(a) $\alpha = 1.0$ $\gamma = 1.0$

Fig. 10: Average steps for winning a game with Q-learning for best values in a 3×3 normal map.

Given the main measure of the performance is the probability of winning a game we establish that the best values for the parameters using Q-Learning in a 3×3 are a learning rate of 1.0, a discount factor of 1.0, and an exploratory factor of 0.02. We have set that the exploratory factor is 0.02 due to the lack disparity between the probability of winning and also because the difference on average steps for winning a game between 0.02 and 0.01 is around 10000 steps more.

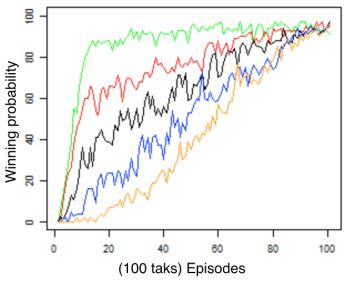
Once we know the best values for a 3×3 map the next step is to use this values for bigger maps than 3×3. The problem with the best values of a 3×3 map is that in a 4×4 gets stuck in a loop, so it never ends due to the infinite reward of going to the ambush zone and coming back to the repair

zone. This loop suggests that the exploratory factor must be increased to make the agent explore more and being able to discover new paths towards the cell that really matters, that is, the attack zone, where our enemy is placed.

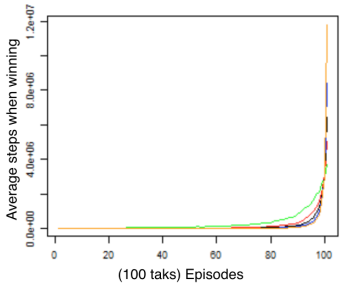
It is worth noting that the increment of the map size leads to an increase in the number of states which entails that the agent needs more time or even a different values of the parameters to be able to explore the new number of states. In particular we have experimented with maps of 4×4 (58 states), 4×4 (100 states), 5×5 (155 states), 6×6 (218 states), 7×7 (295 states) y 8×8 (385 states). With this, we can reach a better understanding of how the exploratory factor may be the crux of the matter for not getting stuck in loops between the repair zone and the ambush zone.

The best values for the parameters in a 3×3 map are not useful for a map of bigger dimensions. We will assume as the baseline that the learning rate and the discount factor are correctly set, and we will modify the value of the exploratory factor that is, indeed, the one that will avoid loops as the one previously mentioned. Besides, we will increase the range of values for the exploratory factor which will pass to be a temporal parameter that will start with a value of x and will finish in the last episode with a value of y, being $x > y$ and diminishing its value progressively.

With the purpose of not spread ourselves too thin, we will start with a 4×4 map. The final value of the exploratory factor is 0.02, pouncing on the fact that this value was the best for a 3×3 map. We will use as initial values for the exploratory factor 0.2, 0.4, 0.6, 0.8 y 1.0. The results can be seen in Figure 11 where we find different aspects of the algorithm's performance for the different configurations. From the one side, we can observe that independently of the initial exploratory factor, the algorithm always converges in the last episodes as soon as the exploratory factor is diminished. In this case, the final value of the exploratory factor is in the last episode, so there is no way to know if the performance will keep at the same height once the value of the parameter is not changing. We couldn't add some episodes for checking this stabilisation of the curve because as you can see in the second plot the number of steps for winning a game increases exponentially what it makes impossible to obtain in a reasonable time conclusions about the convergence of the algorithm.



(a) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.02\}$



(b) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.02\}$

Exploratory Factor	Execution Time
0.2-0.02	17:50:50
0.4-0.02	12:30:55
0.6-0.02	11:03:55
0.8-0.02	10:38:07
1.0-0.02	12:02:42

(c) Execution time for temporal ϵ

Fig. 11: Performance for a 4×4 normal map using Q-Learning and a temporal ϵ-Greedy.

In Figure 11 what wee see from left to right is the probability of winning a game, the number of average steps when winning, and the table of execution times for each configuration using an exploratory factor which starts in the episode 0 with a variable value and finish always in the

episode 100 with a value of 0.02. The size of the map is 4×4. The colour for the initial value of the exploratory factor of 0.2 is green, 0.4 is red, 0.6 is black, 0.8 is blue, and 1.0 for orange. From left to right we see the winning probability, the average steps when winning and the execution time.

Our goal is to achieve the best winning probability. Nevertheless, it's worthless as we can see in Figure 11 to obtain a high winning probability if the time needed for achieving such probability is high too. It's worth noted that what we see in Figure 11 is the average of results obtained when executing 100 games 100 times. This calculation means that the execution time makes reference to the time in executing a total of 10000 games. If we divide the timing by the total number of games, we can calculate roughly speaking that the time for achieving a similar performance would be of 10 minutes which is too much.

Increase the exploratory factor represents at least theoretically that the agent will carry out a lot of errors in the initial games but it will be these errors that will make the agent, at the same time that the value of the exploratory factor is diminished, to reinforce a good state-action policy since there were many visits and estimations of the wide variety of states possibles in earliest episodes. This change in the exploratory factor is due to the results in Figure 11 for avoiding the aforementioned exponential increase in the steps needed for winning a game. We will set as final exploratory factor values 0.05, 0.1, 0.15 y 0.2. The results of this increment can be seen in Figure 12.

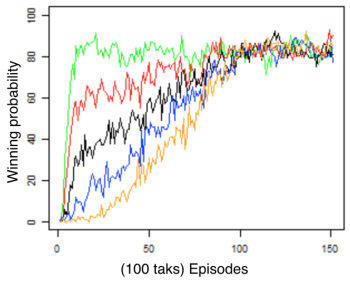
Figure 12 shows from left to right the probability of winning a game using an exploratory factor which starts in the episode 0 with a variable value and finish always in the episode 100 with a variable value (leaving 50 episodes more with a fix configuration of the parameters), the number of average steps when winning, and the table of execution times for each configuration. The size of the map is 4×4. From up to down the final values of the exploratory factor are 0.2, 0.15, 0.1, 0.05 and the initial values are 1.0 for orange, 0.8 for blue, 0.6 for black, 0.4 for red, 0.2 for green. From left to right we see the winning probability and the average steps when winning.

In Figure 12 the last 50 episodes are executed with the last value of the exploratory factor. With this, we can see if there is a stabilisation of the curve in terms of performance. We can observe that in the chart of the probability of winning and the number of steps alike there is such stabilisation. From this results, we can draw that the probability of winning does not depend on the final exploratory value at all since in all cases the performance is around 90%. On the other side, we can see that the execution time is increased as the final value of the exploratory value is lower. For instance, with an initial exploratory factor of 0.2, we would start with a time of one hour for a final value of the exploratory factor of 0.2, two hours for one of 0.15, five hours for one of 0.1, and 22 hours for one of 0.05. A decrease in the final value turns into an increment of the steps needed for winning a game as we can see in the second charts for each row in Figure 12.

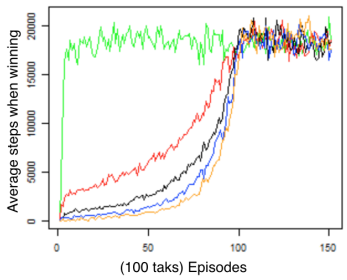
To sum up, we can establish that the number of steps for winning a game is smaller for bigger values of the final value of the exploratory factor, that the performance reaches same heights independently of the final value of the exploratory factor, that the time needed for reach such heights will be lower if the initial exploratory factor is higher, and that the best values for the parameters are a learning rate of 1.0, a discount factor of 1.0, and a temporal exploratory factor *epsilon*-greedy that goes from 1.0 to 0.2.

Testing this last values of the parameters for a 4×4 map we discovered that was impossible to get results for a 5×5 map taking us three days for a progress of just a 50% of the execution. Because of this, we conclude that it is impossible to obtain with Q-Learning a feasible solution for maps greater than 5×5 in a reasonable time.

We finally conclude for this experiment based on the main measure, that is the probability of winning, that the best values for the parameters would be a learning rate of 1.0, a discount factor



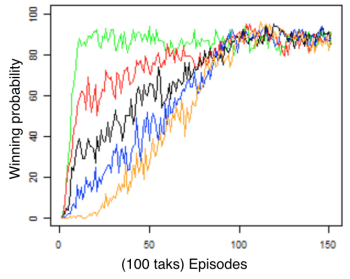
(a) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.2\}$



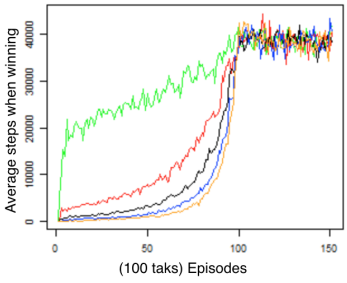
(b) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.2\}$

Exploratory Factor	Execution Time
0.2-0.2	01:21:55
0.4-0.2	01:00:44
0.6-0.2	00:54:31
0.8-0.2	00:49:31
1.0-0.2	00:48:59

(c) Execution time for each range of ϵ



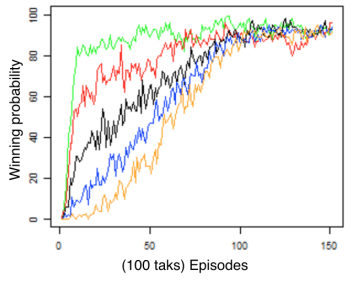
(d) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.15\}$



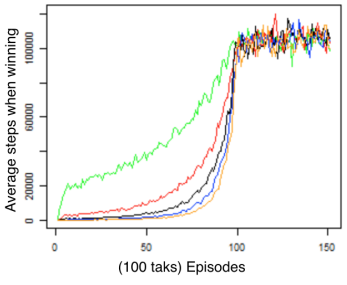
(e) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.15\}$

Exploratory Factor	Execution Time
0.2-0.15	02:28:00
0.4-0.15	01:56:14
0.6-0.15	01:44:22
0.8-0.15	01:40:47
1.0-0.15	01:37:53

(f) Execution time for each range of ϵ



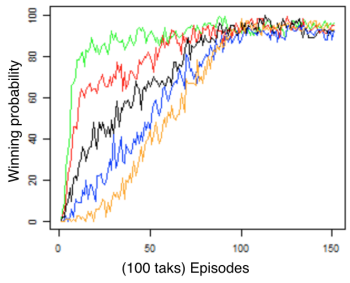
(g) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.1\}$



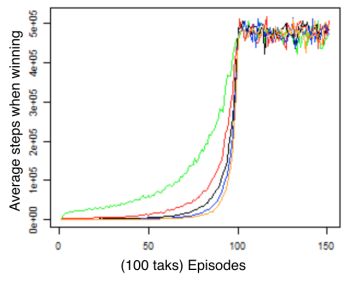
(h) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.1\}$

Exploratory Factor	Execution Time
0.2-0.1	05:28:51
0.4-0.1	04:28:40
0.6-0.1	04:13:44
0.8-0.1	03:56:02
1.0-0.1	03:56:31

(i) Execution time for each range of ϵ



(j) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.05\}$



(k) $\alpha = 1.0\ \gamma = 1.0\ \epsilon = \{v, 0.05\}$

Exploratory Factor	Execution Time
0.2-0.05	21:52:57
0.4-0.05	19:42:11
0.6-0.05	18:38:27
0.8-0.05	17:49:25
1.0-0.05	18:33:09

(l) Execution time for each range of ϵ

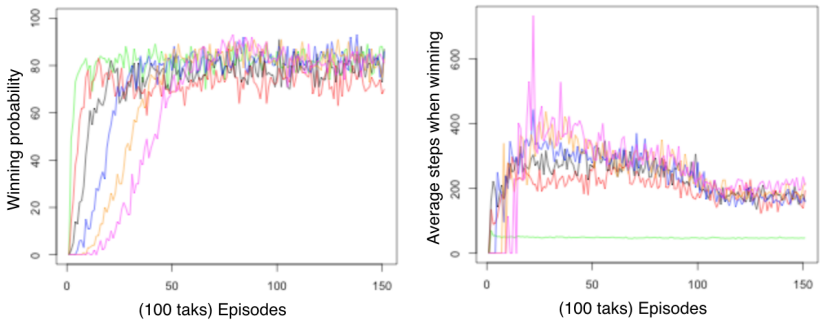
Fig. 12: Performance for a 4×4 normal map using Q-Learning and a temporal ϵ-Greedy with 50 extra episodes for checking convergence.

of 1.0 and an exploratory factor of 0.02 for a 3×3 map and for a 4×4 and 5×5 map the best values are a learning rate of 1.0, a discount factor of 1.0 and a temporal exploratory factor that goes from 1.0 to 0.2.

Experimentation with discrete reward function In this experiment, we will use now the discrete reward function. For that, we have carried out a similar process to which we did for the continuous reward function.

Figure 13 shows the best values for parameters along the different sizes of the map making use of a learning rate of 1.0, a discount factor of 0.98, and a temporal exploratory factor that goes from 0.2 to 0.05 ending up the last 50 episodes with the last value of the exploratory factor affixed. The best values are common to all the maps as of 3×3 maps where only as an exception it comes that for the 3×3 map the best discount factor is 0.0. The green colour corresponds to a 3×3 map, red to a 4×4 map, black to a 5×5 map, blue to a 6×6 map, orange to a 7×7 map, and pink to a 8×8 map. From left to right we see the winning probability and the average steps when winning.

With this reward function, we can see that the probability of winning goes up to 80% in every single size of the game's map. Also, the number of steps goes down considerably taking the average steps when winning of around 250 for every map.



(a) $\alpha = 1.0$ $\gamma = 0.98, \epsilon = \{0.2, 0.05\}$. (b) $\alpha = 1.0$ $\gamma = 0.98, \epsilon = \{0.2, 0.05\}$.

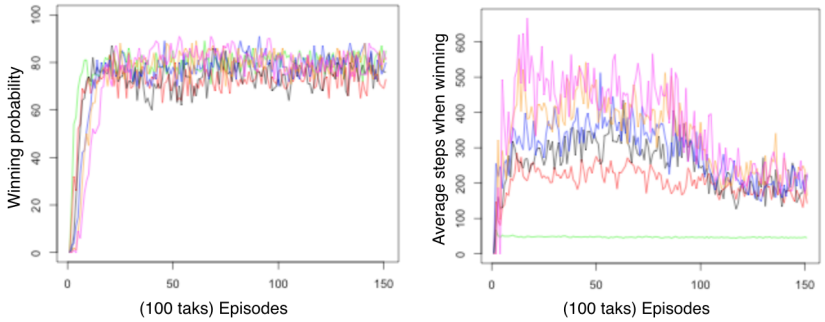
Fig. 13: Best values for normal map using Q-Learning, temporal ϵ-Greedy and a discrete reward function.

The values of the parameters with this discrete reward function does not vary too much with respect to the ones used with the continuous reward function. Indeed, the solely meaningful change has been produced in the start end final value of the exploratory factor, which are bigger in the continuous reward function.

From the results, we can draw a clear conclusion about the importance of the reward function. A suitable reward function to the problem can make the difference in the performance of an algorithm in RL. Additionally, we can finally set that for Q-Learning the best choice is the use of the discrete reward function with a learning rate of 1.0, a discount factor of 0.98, and a temporal exploratory factor that goes from 0.2 to 0.05.

Experimentation with the best configuration of the parameters with Q-Learning in the chess map We describe at the beginning of this section a map Figure 5 so-called chess map that we use to make the agent faces a map for which we have not searched good values for the parameters of the algorithm. In this experiment we do exactly this, we assess how the agent reacts to an environment that has never played before. In Figure 14 we can see the results for the best values of the parameters obtained during the assessment process of Q-Learning in a normal map using such values over a chess map.

Figure 14 shows the best values for the parameters in Q-Learning with different sizes of the chess map. These values are a learning rate of 1.0, a discount factor of 0.98, and a temporal exploratory factor that goes from 0.2 to 0.05 executing a total of 150 episodes where the last 50 episodes have the values affixed so $\epsilon = 0.05$. The green colour corresponds to a 3×3 map, red to a 4×4 map, black to a 5×5 map, blue to a 6×6 map, orange to a 7×7 map, and pink to a 8×8 map. From left to right we see the winning probability and the average steps when winning.



(a) $\alpha = 1.0$ $\gamma = 0.98$ $\epsilon = \{0.2, 0.05\}$. (b) $\alpha = 1.0$ $\gamma = 0.98$ $\epsilon = \{0.2, 0.05\}$.

Fig. 14: Best values for chess map using Q-Learning, temporal ϵ-Greedy and a discrete reward function.

The results obtained are extremely good except for some smooth variations in the number of steps needed when winning a game with respect to the normal map, where there is a bigger and faster convergence. With respect to the main measure, the probability of winning, the algorithms keep high around a 80% as in the normal map with the improvement of not needing less time of adaptation in the first episodes. This last points out that the chess map is easier for the agent. This is probably due to the bigger number of paths that exists between the agent and the enemy as opposite in the normal map where there are only two paths.

5.2. SARSA

Description SARSA is an on-policy difference temporal method. An on-policy method is those that learns the values of a policy that is being used for taking decisions. Common policies are softmax or ϵ-greedy. These policies make sure that always there is an element of exploration and therefore include a non-deterministic factor in the algorithm. The main difference among SARSA

and Q-Learning is that not necessarily the maximum reward of the next state in the policy is used for updating the Q-values.

SARSA is an algorithm that makes reference to the initials State-Action-Reward-State-Action. These initials makes reference to the way in which SARSA carries out the updating of the Q-values following the form $Q(s, a, r, s', a')$ where s and a are the old state and action, r is the observed reward for the new state, s' and a' are the new state and action based on the current policy. The process to follow is similar to Q-Learning such as we can see in Listing 2, setting just a subtle difference in the calculus of the estimations and in the extra phase of action selection.

```
Initialize Q(s,a) // (Initially 0)
REPEAT(nEpisodes):
    Initialize s
  Choose a for s using Q
  REPEAT(until s is terminal):
      Execute a, observe the new r y s'
    Choose a' for s' using Q
    Q(s,a) = Q(s,a) + α[r + γ * Q(s',a') − Q(s,a)]
    s = s'
    a = a'
  END_REPEAT
END_REPEAT
```

Code 2: SARSA.

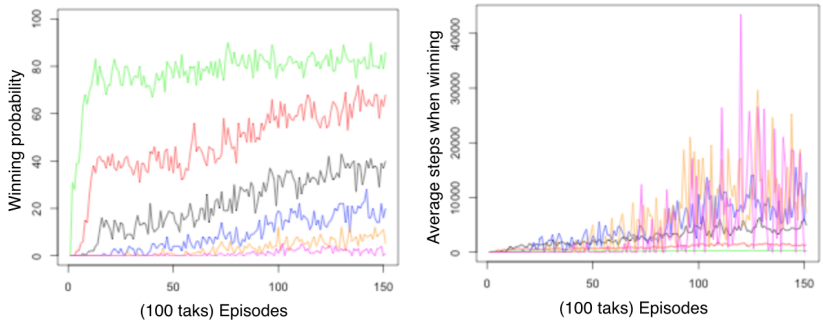
Experimentation with continuous reward function We have used for SARSA the same values than in 5.1. Indeed the same problems occurred with respect to the rise of average steps needed for winning a game, which was resolved by increasing the final exploratory factor.

As a result of the same strategy followed in 5.1. Q-Learning we obtain as results Figure 15 where we see better results than those obtained with Q-Learning could even to carry out executions for maps up to size 8 without the need of changing the reward function, which was necessary to do it with Q-Learning in order to obtain results for big maps.

Figure 15 shows the use of a temporal exploratory factor that goes from 0.2 to 0.01 executing a total of 100 episodes over the different sizes of the normal map. The color for a 3×3 map is green, red for a 4×4 map, black for a 5×5 map, blue for a 6×6 map, orange for a 7×7 map, pink for a 8×8 map. From left to right we see the winning probability and the average steps when winning.

We can draw from the experiments with the on-policy method SARSA that the optimal values for the parameters are a learning rate of 1.0, a discount factor of 0.7, and a temporal exploratory factor that goes from 0.2 to 0.05. SARSA has an association with its estimation due to its relation to the usage of the next state and the next possible action to be used what we think have got a straight result in the efficiency of one of the two optimal paths that lead to the enemy in the normal map. SARSA creates because of its way to estimates the values a breadcrumbs path that ease to get faster to the place where the enemy is. This bread crumbs path is a trace of positive Q-values that goes through the Q-table that leads to the enemy cell.

Experimentation with the discrete reward function A similar process for getting the best values of the parameters has been done in this experiment as we already did in 5.1. Q-Learning for the different sizes of the map with the continuous reward function.



(a) $\alpha = 1.0$ $\gamma = 0.7$ $\epsilon = \{0.2, 0.05\}$ (b) $\alpha = 1.0$ $\gamma = 0.7$ $\epsilon = \{0.2, 0.05\}$

Fig. 15: Best values for normal map using SARSA, ϵ-Greedy and continuous reward function.

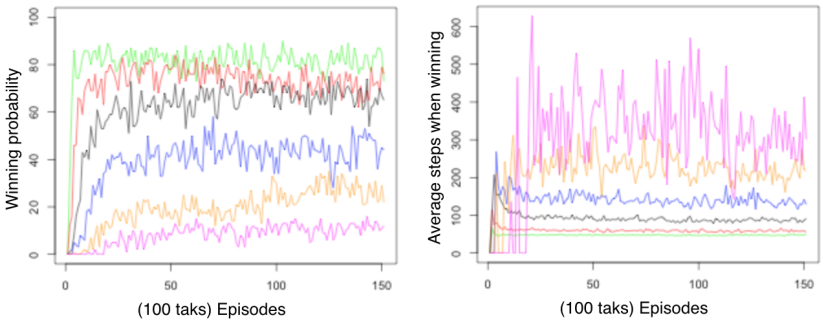
The results of this process can be seen in Figure 16 that shows the best values for the parameters with different sizes of the map having a learning rate of 1.0, a discount factor of 0.2, and a temporal exploratory factor that goes from 0.1 to 0.05 executing a total of 150 with the last 50 episodes with the final values of the parameters. The color for a 3×3 map is green, red for a 4×4 map, black for a 5×5 map, blue for a 6×6 map, orange for a 7×7 map, pink for a 8×8 map.

We can see in Figure 16 the best values for the different sizes of the map with the discrete reward function. Whereas for the continuous reward function the best values were a learning rate of 1.0, a discount factor of 0.7, and a temporal exploratory factor that goes from 0.2 to 0.05, for the discrete reward function we have a learning rate of 1.0, a discount factor of 0.2, and a temporal exploratory factor that goes from 0.1 to 0.05. There exist an exception for 3×3 map where the best discount factor is 0.0 and not 0.2. From left to right we see the winning probability and the average steps when winning.

In Figure 16 we can see that unlike Figure 15 the average number of steps for winning a game are reduced drastically until reaching a maximum number of 600, whereas with the continuous reward function this number goes up exponentially for a small exploratory factor. Besides, the winning probability of winning a game grows up in general for all the sizes of the map making SARSA a most suitable solution than Q-Learning up to now.

The most meaningful change in the values of the parameters for the discrete reward function with respect to the continuous reward function is the discount factor. Remind that the discount factor has got as main goal to give more or less importance to the reward in long-term. Since the reward function provides discrete and exact rewards for each zone of the map, there is no such a huge need for looking a big reward in long-term and it is because of this that the value of the parameter is diminished.

Experimentation with action selection methods UCB and soft-max SARSA is an on-policy method what means that the results of this method are extremely related to the action selection method that is using. Taking this into account and once we have discovered which of our reward function is better, that is, the discrete reward function, we are going to focus our attention on the action selection method. We will check which are our results with the action selection methods UCB and soft-max.



(a) $\alpha = 1.0$ $\gamma = 0.2$ $\epsilon = \{0.2, 0.05\}$. (b) $\alpha = 1.0$ $\gamma = 0.2$ $\epsilon = \{0.2, 0.05\}$.

Fig. 16: Best values for normal map using SARSA, ϵ-Greedy and discrete reward function.

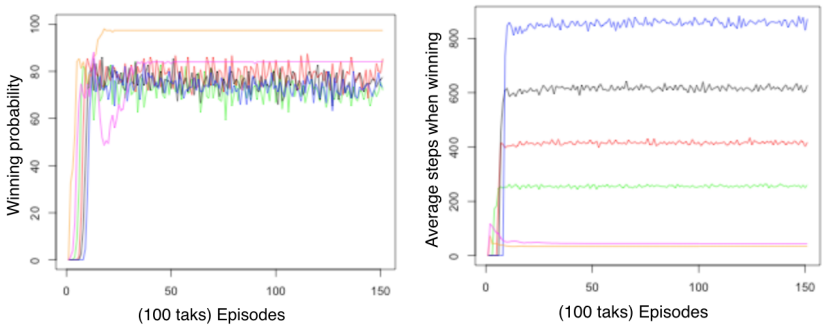
UCB (Upper-Confident-Bound) carries out the action selection, as opposite to ϵ-greedy, taking into account how close an estimation is near to the optimal value as well as the uncertainty of such estimation. The way of doing this is using Equation 5 where $log(t)$ denotes the natural logarithm of the time t, $N_t(a)$ is the number of times that action a has been selected before time t, and the number $c > 0$ is the one that controls the degree of exploration of the algorithm. If $N_t(a) = 0$ then this action is considered maximising, that is, the action would be selected among the others maximising actions, if there exist any other at time t.

$$A_t = \max_a \left[Q_t(a) + c\sqrt{\frac{log(t)}{N_t(a)}} \right] \tag{5}$$

Figure 17 shows best values for the parameters with SARSA, UCB, and a discrete function reward making use of a learning rate of 1.0, a discount factor of 0.0, and an exploratory factor $c = 0.01$ executing a total of 150 episodes over the different sizes of the map. For a map of sizes 3 and 4, the best discount factor is 0.6. The color for a 3×3 map is orange, pink for a 4×4 map, green for a 5×5 map, red for a 6×6 map, black for a 7×7 map, blue for a 8×8 map. From left to right we see the winning probability and the average steps when winning.

In Figure 17 we can see that the more informed search of UCB about which actions must be taken based on the uncertainty of these actions and its possible maximising factor makes UCB a better alternative than ϵ-Greedy as action selection method with SARSA. The results are around 75% for the different sizes of the map and the average of steps needed when winning a game go up progressively in a softly way without huge jumps as we have seen before.

With respect to the second action selection method used with SARSA, we have soft-max. Soft-max is an action selection method which estimates the Q-values using probabilities. To each action, a probability of how probable this action should be chosen is assigned. This probability sets a preference of selection from one action to another. For the implementation of soft-max, we have made used of Boltzmann distribution which assigns the probabilities or preferences using Equation 6 for each action in time t and using a so-called temperature τ parameter that indicates how close the algorithm should behave as the ϵ-Greedy does. A value close to zero will make soft-max be-



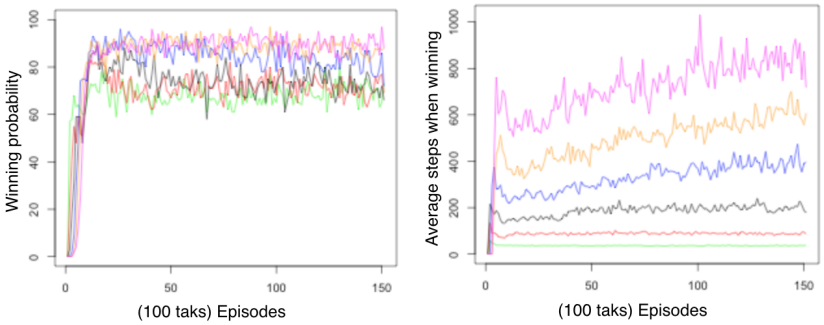
(a) $\alpha = 1.0$ $\gamma = 0.0$ $c = 0.01$. (b) $\alpha = 1.0$ $\gamma = 0.0$ $c = 0.01$.

Fig. 17: Best values for normal map using SARSA, UCB and discrete reward function.

haves almost as the ϵ-Greedy method whereas higher values will make the different actions more equiprobable to each other.

$$Pr(A_t) = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{k} e^{Q_t(b)/\tau}} \tag{6}$$

Once we have assigned the probabilities to each action, we set a method for selecting the actions based on these probabilities. For this we have used a well-known algorithm for the selection of individuals in genetic algorithms as it is the roulette wheel selection method [39] [40]. This algorithm makes use of an accumulative distribution function (adf) of the calculated probabilities and a parameter r that is a random value which sets the action that will be executed depending on which range of the adf is.



(a) $\alpha = 1.0$ $\gamma = 0.6$ $\tau = 0.2$. (b) $\alpha = 1.0$ $\gamma = 0.6$ $\tau = 0.2$.

Fig. 18: Best values for normal map using SARSA, soft-max and discrete reward function.

Figure 18 shows the best values for the parameters with SARSA, soft-max, and a discrete function reward in a normal map making use of a learning rate of 1.0, a discount factor of 0.6, and an exploratory factor so-called temperature in soft-max with $\tau = 0.01$ executing a total of 100 episodes over the different sizes of the map. For maps of sizes 3 and 4, the best discount factor is 0.6. For a map of sizes 3 and 4, the best discount factor is 0.6. The color for a 3×3 map is green, red for a 4×4 map, black for a 5×5 map, blue for a 6×6 map, orange for a 7×7 map, pink for a 8×8 map. From left to right we see the winning probability, and the average steps when winning.

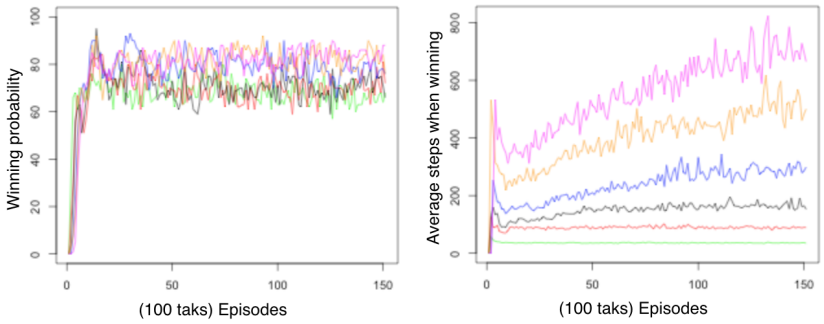
In Figure 18 we can see the results of SARSA making use of the action selection method soft-max. What we see in the plots is that soft-max is able to keep a probability gain higher for the different sizes of the map without increase exponentially the average of steps needed for winning a game as we saw with the temporal ϵ-Greedy algorithm. Indeed, the results of soft-max are even better than the ones with UCB with the difference that UCB seems to keep a better stabilisation of the average steps needed when winning. Soft-max achieves overall to be around a 85% of the probability of winning, that is, better than with UCB.

There is no theory about which action selection is better than other but just a claiming that depending on our problem one method will be better than others. In our case, the results are quite similar, so we will not take a risk making hasty conclusions but just drawing the fact that it seems that the estimations based on probabilities that soft-max does fit better with our problem than the estimates made by UCB without having a significant difference among them.

Experimentation with the best alternative of the SARSA method with the normal map in the chess map Finally, as we said at the beginning of this section, we will face the agent with the best values of the parameters for SARSA to a different map so-called chess map that we can in Figure 5. Just worth noted that we haven't looked for good values of SARSA for this map. The goal of this experiment, as an opposite to the rest that we have carried out, is to check only the behaviour of the algorithm when playing in a new map from which it was firstly searched the best values of the parameters.

Figure 19 shows the best values for the parameters with SARSA, soft-max, and a discrete reward function in a chess map making use of a learning rate of 1.0, a discount factor of 0.6, and an exploratory factor so-called temperature in soft-max with $\tau = 0.2$ executing a total of 100 episodes over the different sizes of the map. For a map of sizes 3 and 4, the best discount factor is 0.6. The color for a 3×3 map is green, red for a 4×4 map, black for a 5×5 map, blue for a 6×6 map, orange for a 7×7 map, pink for a 8×8 map. From left to right we see the winning probability, and the average steps when winning.

In Figure 19 we can see that despite the abrupt change of the map, SARSA combined with soft-max is able to keep a good performance in terms of winning probability and even better in the average steps when winning a game. If we compare these results with the ones obtained with Q-Learning what we see is that Q-Learning is better since it obtained a similar performance to SARSA but with the difference that once the convergence is reached Q-Learning needs fewer steps for winning a game than SARSA. This result has got a solid theoretical basis, and it is that whereas SARSA using soft-max needs test at least once all the possible actions, Q-Learning did not have to do this since the action selection method used with this learning method was ϵ-Greedy what drives into an inferior number of random steps.



(a) $\alpha = 1.0$ $\gamma = 0.6$ $\tau = 0.2$. (b) $\alpha = 1.0$ $\gamma = 0.6$ $\tau = 0.2$.

Fig. 19: Best values for chess map using SARSA, soft-max and discrete reward function.

5.3. Monte Carlo Tree Search & Q-Learning

Monte Carlo Tree Search (MCTS) is an action selection method used for Markov Decision Process (MDP) problems where there exist a large space of states [41]. Nowadays the best AI have got implemented a MCTS, prove of it is its use in board games with branch factors of 275 as Go [42] or real-time games as Tron [43].

The MCTS algorithm consists of the creation of a tree which has four phases: selection, expansion, simulation and backpropagation. These four phases are executing once and again as we see in Figure 20 until we decide when stopping such execution, whether that be a maximum number of repetitions, an execution time or a minimum threshold of the tree's entropy.

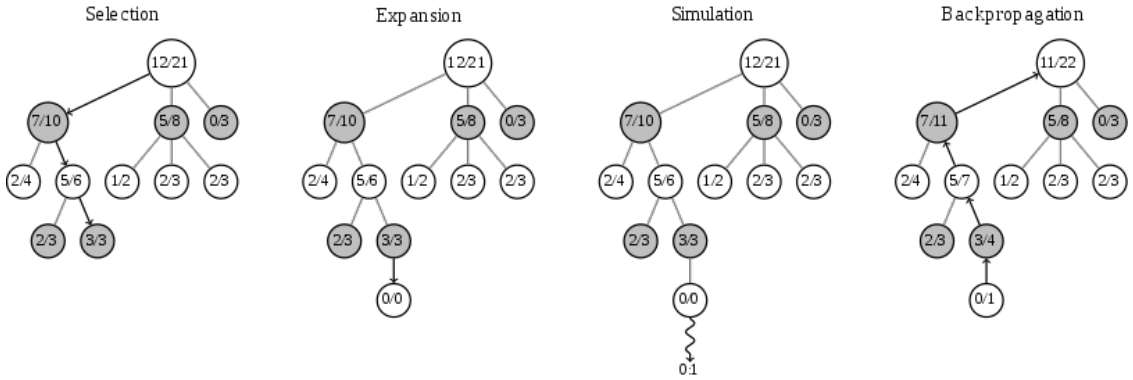


Fig. 20: Phases of the execution of a Monte Carlo Tree Search.

The first phase of the tree is the selection of a node. Such selection is made through the UCB (Upper Confidence Bound) algorithm [44]. The variant of this algorithm for trees is UCT. Provided that the children nodes have been selected at least once, the UCT algorithm will come into the selection process selecting a child based on its statistical data. UCT selects a child node k from the

set of children K of a particular father node using Equation 7. In Equation 7 η_i is the number of visits to the node child i whereas X_i is the average reward of the node child i. The parameter C carries out the exploration function, and it is the solely available parameter for fitting the algorithm performance.

$$k = \max_{i \in K} \left(X_i + C \sqrt{\frac{\ln(\eta_j)}{\eta_i}} \right) \tag{7}$$

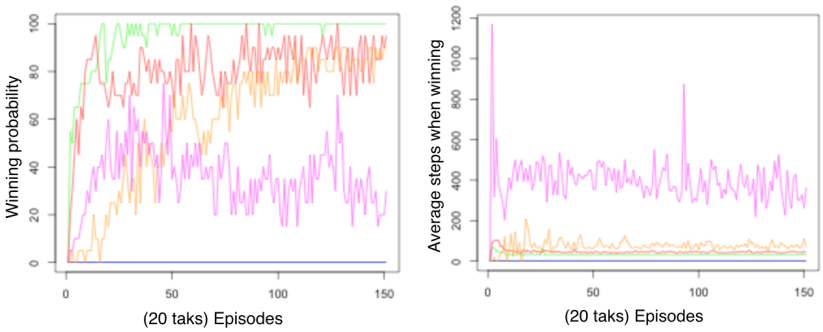
Once we have selected a node using the UCT algorithm, we would pass to the expansion phase. In the expansion phase, the node mentioned above selected is expanded creating the children of this node based on the possible actions to be taken from the state represented by the node selected. This set of actions is dependent on the model we have implemented since it's worth noted that for the use of a MCTS it is necessary to have implemented a model of the environment, whether that be the real model or a simulated model of the real environment, which the agent will face.

When the selected node is expanded, we pass to the simulation part. In this phase using the implemented model and based on a default policy which is usually to take random actions, the agent has to reach a terminal state. Finally, we go to the last phase, that is, backpropagation in which using the reward acquired when reaching a terminal state we propagate such reward along the branches of the expanded node selected through the top until we get to the root node of the tree. Along the backpropagation, the statistical data of the nodes that got involved in the path up to the terminal state are updated.

For the experimentation with this algorithm, we have carried out the simplest version. In particular, we have carried out the creation of the tree for each movement, that is, we didn't reuse any part of older trees that could have been of help for next movements. Besides, we implemented initially a default policy based on random movements when being in the simulation phase. Since the goal of this project is to acquire how much knowledge of RL as possible we finally set as the policy of the MCTS the action selection method ϵ-Greedy that uses Q-values of the Q-table that is updated through the use of the Q-Learning algorithm. We have established the parameter C with the final value of 1 for delimiting the range of parameters to be set. The best results can be seen in Figure 21.

Figure 21 shows the best values for the parameters with MCTS, using as internal policy Q-Learning and ϵ-Greedy in a normal map making use of a learning rate of 1.0, a discount factor of 0.98, an exploratory factor $\epsilon = 0.2$, a number of $\beta = 100$ MCTS simulations, taking as many steps for finishing the execution (this is marked with a $\theta = 0$) and executing a total of 150 episodes over the different sizes of the map. For a map of sizes 5 and 6, the best number of simulations is a $\beta = 200$ and a $\theta = 0$, respectively. The color for a 3×3 map is green, red for a 4×4 map, orange for a 5×5 map, pink for a 6×6 map, black for a 7×7 map, blue for a 8×8 map. From left to right we see the winning probability, and the average steps when winning.

The results obtained with MCTS are acceptable for maps of size 3, 4 and 5. As opposite, for maps of sizes 6, 7 and 8 the results are rather disastrous. The result of this diminishing in the performance is the product of the dependency of the MCTS and the Q-Table of the Q-Learning algorithm. Such dependency is bigger as the size of the map increase due to the increase in the number of states in the table and therefore the time needed for estimating the values of the pairs state-action are bigger. The actions taken during the simulation phase of the MCTS will be as good as the Q-Table is. Besides, we didn't do a fit MCTS so factors as the re-creation of the tree for each action plays against the performance of the algorithm leading to a high computational cost and



(a) $\alpha = 1.0$ $\gamma = 0.98$ $\beta = 100$ $\theta = 0.0$ $\epsilon = 0.2$.

(b) $\alpha = 1.0$ $\gamma = 0.98$ $\beta = 100$ $\theta = 0.0$ $\epsilon = 0.2$.

Fig. 21: Best values for normal map using MCTS, Q-Learning, ϵ-Greedy and discrete reward function.

producing long-time executions. This issues would be solved if we stored and reused the trees we created making use of its knowledge in each episode. Based on the results we can draw that if we would leave the execution for a time bigger than 150 episodes the performance in terms of winning probability of the maps of sizes 6, 7 and 8 will be presumably equal to the maps of sizes 3, 4 and 5.

As a characteristic to be highlighted of the MCTS is its capacity in terms of stabilisation between exploration and exploitation due to the results in terms of average number of steps when winning a game as we can see in Figure 21 (right plot) where the number is less than 200 steps for maps of sizes 3, 4 and 5. This advantage of MCTS is something logical as the model in the MCTS is exactly the same that the agent faces so, presumably, if we made some improvements in MCTS code we could as good results as nearly 100% winning probability in a short period of time as we can see for the 3×3 map.

6. Global Analysis

In this section, we will make a global analysis of the different solutions given to the problem. For that, we have selected the 6×6 map as the yardstick of the different sizes for having enough number of states for evaluating the performance ruling out the possibility of winning a game for just random lucky actions. We have used the best values of the parameters for each algorithm we have developed and the results can be seen in Figure 22.

Figure 22 shows the results for a 6×6 normal map of the best solutions for the different algorithms developed in this project: Q-Learning, SARSA, and MCTS. SARSA is the black curve, Q-Learning is green one, and MCTS is the red one. From left to right we see the winning probability, the average steps when winning, and the average reward. Q-Learning with ϵ-Greedy uses $\alpha = 1.0$, $\gamma = 0.98$, and a temporal ϵ that goes from 0.2 to 0.05. Q-Learning with MCTS uses $\alpha = 1.0$, $\gamma = 0.98$, and a temporal $\epsilon = 0.2$, $\beta = 200$, and $\theta = 20$. SARSA uses $\alpha = 1.0$, $\gamma = 0.6$, and $\tau = 0.2$.

The three algorithms give approximately the same average number of steps when winning a game. Nevertheless, the probability of winning of Q-Learning and SARSA surpass the performance of the MCTS algorithm. This is partially due to fact that the implementation of the MCTS algorithm

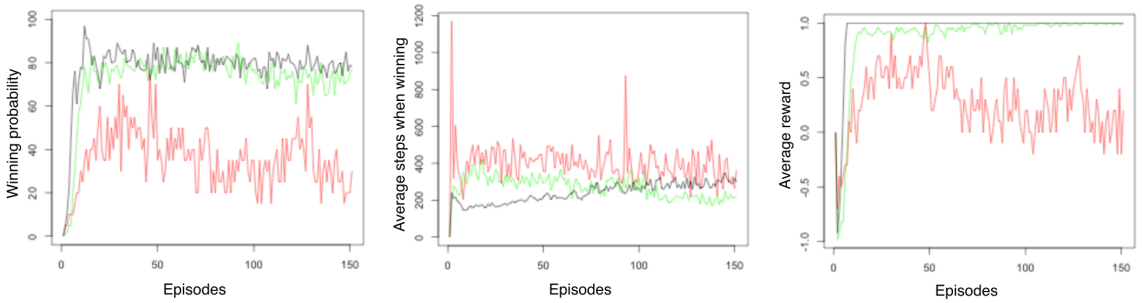


Fig. 22: Results for a 6×6 normal map for MCTS, SARSA, and Q-Learning.

is in its basis without introducing any further detail than the connection between the Q-Learning algorithm and the MCTS algorithm throughout the Q-Table as we said in the subsection of the MCTS algorithm in 5. Solution.

From the results we can draw that SARSA is the algorithm that fits better to our problem as even though the results in Figure 22 are practically equals to the Q-Learning's results, we do observe a disparity in the average reward which SARSA keeps stabilized much better than Q-Learning, which is more affected by this measure due to the random framework attached to the action selection method ϵ-Greedy.

7. Conclusions and future work

A study of the different algorithms in RL has been carried out implementing those considered to be first priority or what we have named as classics. The developed algorithms have been both learning methods and action selection methods. We have reached the final conclusion that the algorithms that best fit our problem is the learning method SARSA along with the action selection method soft-max.

Throughout the development of this project, we have concluded from the experience gained that there does not exist such a thing as the best learning method or the best action selection method, already stated by Sutton & Barto in [1]. All depending on our problem, one method will be better than the other. As we saw, not only the methods are relevant for the good performance of a RL algorithm, but also the reward function and the values of the parameters in these methods. As we saw with Q-Learning, some executions took hours to finish when we only needed to increment the final value of the exploratory factor in order to make it faster and at a performance of around 80%. From the whole development of this project, if there is something we can conclude, it is that a good reward function might be the crux of the matter. We only needed to change the reward function from the continuous to the discrete reward function for moving from a poor performance and a huge amount of time for executions to a reasonable time in executions and performance.

On the other hand, we have the representation of the knowledge itself. In this project, we have set this representation to a simple table where the x-axes are the states and the y-axes are the actions. We encountered issues with this representation of the knowledge as soon as the number of states was increased. This leads us to think that, based on the experience acquired in this project, a simple table may not the best way to deal with game states as these increase rapidly.

Additionally, we included in this project the Monte Carlo Tree Search as a method for the planning phase in RL algorithms. This planning phase method widely used in video games did not match our expectations. The naive implementation of the MCTS algorithm was not enough to achieve a similar performance to other algorithms in our video game. In any case, we quickly realised that this method, despite working just with basic information about the game, it would actually need some injection of knowledge in order to get the best out of it.

Points that have not been dealt with in this dissertation, however, which will be approached in a further work study are the improvement of the MCTS algorithm as well as the integration of artificial neural networks with the RL. This last computational model has indeed been developed however due to the limitations in the length of this document, we have omitted it, not to mention the fact about the complexity of designing an artificial neural network and the time it takes to fit this computational model to a specific problem.

The knowledge acquired during this dissertation is meant to be a boost towards the RL research field and its wider context. Knowing the fundamentals of a field is the first step towards the research of it, and it is due to this that we have focused our efforts towards this goal.

References

[1] Barto Sutton Richard S. and Andrew G. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT Press Cambridge, 1998.

[2] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-art*. Vol. 12. Springer Science & Business Media, 2012.

[3] Gerald Tesauro. «Programming backgammon using self-teaching neural nets». In: *Artificial Intelligence* 134.1 (2002), pp. 181–199.

[4] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529.7587 (2016), pp. 484–489.

[5] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *Nature* 518.7540 (2015), pp. 529–533.

[6] Harm V Seijen and Rich Sutton. «True Online TD (lambda)». In: *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 2014, pp. 692–700.

[7] Yann A LeCun et al. «Efficient backprop». In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[8] Thore Graepel, Ralf Herbrich, and Julian Gold. «Learning to fight». In: *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*. 2004, pp. 193–200.

[9] Guillaume Chaslot et al. «Monte-Carlo Tree Search: A New Framework for Game AI.» In: *AIIDE*. 2008.

[10] K. Hamahata et al. «Effective Integration of Imitation Learning and Reinforcement Learning by Generating Internal Reward». In: *Intelligent Systems Design and Applications, 2008. ISDA '08. Eighth International Conference on*. Vol. 3. 2008, pp. 121–126. DOI: 10.1109/ISDA.2008.325.

[11] Arthur L Samuel. «Some studies in machine learning using the game of checkers». In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[12] Arthur L Samuel. «Some studies in machine learning using the game of checkers. II—recent progress». In: *IBM Journal of research and development* 11.6 (1967), pp. 601–617.

[13] G. Boone. «Efficient reinforcement learning: model-based Acrobot control». In: *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*. Vol. 1. Apr. 1997, 229–234 vol.1. DOI: 10.1109/ROBOT.1997.620043.

[14] Robert Crites and Andrew Barto. «Improving Elevator Performance Using Reinforcement Learning». In: *Advances in Neural Information Processing Systems 8*. MIT Press, 1996, pp. 1017–1023.

[15] Satinder Singh and Dimitri Bertsekas. «Reinforcement learning for dynamic channel allocation in cellular telephone systems». In: *Advances in neural information processing systems* (1997), pp. 974–980.

[16] L. Busoniu and B. Babuska R. De Schutter. «A Comprehensive Survey of Multiagent Reinforcement Learning». In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 38.2 (Feb. 2008), pp. 156–172. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2007.913919.

[17] Vincent Conitzer and Tuomas Sandholm. «AWESOME: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents». In: *Machine Learning* 67.1-2 (2007), pp. 23–43.

[18] Michael Bowling and Manuela Veloso. «Multiagent learning using a variable learning rate». In: *Artificial Intelligence* 136.2 (2002), pp. 215–250.

[19] International Proyect. *RoboCup Official Website*. 1997. URL: http://www.robocup.org/ (visited on 08/30/2016).

[20] E. Uchibe and M. Asada. «Multiple reward criterion for cooperative behavior acquisition in a multiagent environment». In: *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on*. Vol. 6. 1999, 710–715 vol.6. DOI: 10.1109/ICSMC.1999.816638.

[21] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *Nature* 518.7540 (2015), pp. 529–533.

[22] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529.7587 (2016), pp. 484–489.

[23] Alex J Champandard. «Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI». In: *AIGameDev. com: http://aigamedev. com/open/coverage/mcts-rome-ii* (2014).

[24] Rémi Coulom et al. «Monte-Carlo Tree Search in Crazy Stone,"» in: *Proc. Game Prog. Workshop, Tokyo, Japan*. 2007, pp. 74–75.

[25] Levente Kocsis and Csaba Szepesvári. «Bandit based Monte-Carlo Planning». In: *In: ECML-06. Number 4212 in LNCS*. Springer, 2006, pp. 282–293.

[26] Edward L Thorndike. «The law of effect». In: *The American Journal of Psychology* 39.1/4 (1927), pp. 212–222.

[27] RS Woodworth and H Schlosberg. «Experimental psychology. New York: Henry Holt and Company». In: (1938).

[28] Clark Hull. «Principles of behavior». In: (1943).

[29] BF Skinner. «The behavior of organisms: an experimental analysis. Appleton-Century». In: *New York* (1938).

[30] Christopher D Adams and Anthony Dickinson. «Instrumental responding following reinforcer devaluation». In: *The Quarterly journal of experimental psychology* 33.2 (1981), pp. 109–121.

[31] Nathaniel D Daw, Yael Niv, and Peter Dayan. «Uncertainty-based competition between prefrontal and dorsolateral striatal systems for behavioral control». In: *Nature neuroscience* 8.12 (2005), pp. 1704–1711.

[32] Robert A Rescorla, Allan R Wagner, et al. «A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement». In: *Classical conditioning II: Current research and theory* 2 (1972), pp. 64–99.

[33] Oracle. *Java Official Website*. 2016. URL: `https://www.oracle.com/es/java/index.html` (visited on 08/30/2016).

[34] Eclipse. *AspectJ Official Website*. 2016. URL: `https://eclipse.org/aspectj/` (visited on 08/30/2016).

[35] Gregor Kiczales et al. «An overview of AspectJ». In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 327–354.

[36] Robert Gentleman et al. *R Official Website*. 2016. URL: `https://www.r-project.org/` (visited on 08/30/2016).

[37] Simon Urbanek. *Rserve Official Website*. 2016. URL: `https://rforge.net/Rserve/` (visited on 08/30/2016).

[38] Ben Fry and Casey Reas. *Processing Official Website*. 2016. URL: `https://processing.org/` (visited on 08/30/2016).

[39] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[40] Lourdes Araujo and Carlos Cervigón. *Algoritmos evolutivos: un enfoque práctico*. Alfaomega, 2009.

[41] Levente Kocsis and Csaba Szepesvári. «Bandit Based Monte-Carlo Planning». In: *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-46056-5. DOI: `10.1007/11871842_29`. URL: `http://dx.doi.org/10.1007/11871842_29`.

[42] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529.7587 (2016), pp. 484–489.

[43] Marc Lanctot et al. «Monte Carlo tree search for simultaneous move games: A case study in the game of Tron». In: *BNAIC 2013: Proceedings of the 25th Benelux Conference on Artificial Intelligence, Delft, The Netherlands, November 7-8, 2013*. Delft University of Technology (TU Delft); under the auspices of the Benelux Association for Artificial Intelligence (BNVKI), the Dutch Research School for Information, and Knowledge Systems (SIKS). 2013.

[44] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. «Finite-time analysis of the multiarmed bandit problem». In: *Machine learning* 47.2-3 (2002), pp. 235–256.