Resolución de laberintos en *StarCraft* empleando aprendizaje por refuerzo

Realizado por:

Juan Deltell Mendicute

Alberto Lorente Sánchez

Jesús Martínez Dotor

Trabajo de fin de grado del Grado en Ingeniería del Software

Facultad de Informática

Universidad Complutense de Madrid

Curso 2014/2015

Director:

Antonio Sánchez Ruiz-Granados

Departamento de Ingeniería de Software e Inteligencia Artificial

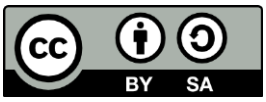
Este documento está preparado para ser imprimido a doble cara.

Autorización de difusión y utilización

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Juan Deltell Mendicute Alberto Lorente Sánchez Jesús Martínez Dotor

Madrid, 19 de junio de 2015



El código desarrollado se encuentra en el siguiente repositorio público de *GitHub* [2]:

https://github.com/TFG-RL-*StarCraft*/TFG*StarCraft*

Dicho código se publica bajo una licencia libre GPLv3: GNU *General Public License* versión 3 [3].

A nuestras familias y novias,
por aguantarnos.

Índice

Índice de figuras

Resumen

Desde los orígenes de la informática se ha intentado dotar de mayor autonomía a los sistemas informáticos, desarrollando algoritmos capaces de aprender y adaptarse a los cambios en su entorno. En el campo de los videojuegos, esta inteligencia artificial (IA) suele quedar limitada a comportamientos previamente definidos por un diseñador, dando lugar a jugadores no humanos carentes de toda autonomía.

Por otro lado, el aprendizaje automático y los algoritmos de aprendizaje por refuerzo son técnicas prometedoras que nos ofrecen la posibilidad de crear agentes que aprendan por sí mismos. Además, a pesar de que se están empezando a introducir en el desarrollo de videojuegos, aún no han sido ampliamente exploradas en este campo. En concreto, el algoritmo *Q-Learning* nos ofrece una buena base para desarrollar sistemas basados en este tipo de aprendizaje.

En este proyecto intentaremos desarrollar una aplicación que mediante este algoritmo sea capaz de aprender a hacer ciertas actividades por sí mismo en el entorno del juego de estrategia en tiempo real (RTS, del inglés *Real-Time Strategy Games*) *StarCraft*. En concreto, lo emplearemos para la resolución de laberintos de forma autónoma, es decir, sin proporcionarle conocimiento previo al agente sobre la estructura del laberinto. Para ello iremos moviendo a una unidad del juego por un laberinto que podrá contener trampas, y mediante aprendizaje por refuerzo lograremos que dicha unidad sea capaz de encontrar la salida por sí misma.

Mostraremos experimentos con distintas estrategias de aprendizaje, usando distintas recompensas, y al final explicaremos cuáles han funcionado mejor. Además desarrollaremos un *framework* reutilizable para este tipo de aprendizaje basado en el algoritmo *Q-Learning*, que emplearemos para resolver nuestro problema y podrá servir como base para futuros desarrollos.

Palabras Clave: aprendizaje por refuerzo, *Q-Learning*, *framework*, *StarCraft*, juegos de estrategia en tiempo real, laberinto.

Abstract

From the beginnings of computer science it has been tried to give more autonomy to computer systems, developing algorithms able to learn and adapt themselves to the changes in their environment. In the area of videogames, this artificial intelligence (AI) is usually limited to behaviors that have been defined in advance by the designer, leading to non-player characters without autonomy.

On the other hand, the machine learning and reinforcement learning algorithms are promising techniques, which give the opportunity to create agents who can learn by themselves. In addition, despite the fact that they are beginning to be introduced in the videogames development, they haven't been widely explored in this area yet. Concretely, the *Q-Learning* algorithm gives us a good base to develop systems based on this learning model.

With this project we will try to develop an application that, using the *Q-Learning* algorithm, will be able to learn several activities by itself in the environment of real-time strategy game (RTS) *StarCraft*. In particular, we will use it to solve maze on an autonomous way, in other words, without providing it with previous knowledge about the maze's structure. To accomplish it, we will be moving one game-unity inside a maze that may have enemies, and thanks to the reinforcement learning the game-unity will be able to find the maze's exit by itself.

We will show experiments with different learning strategies, using diverse rewards, and at the end we will explain which of them worked better. Also we will develop a reusable *framework* for this kind of learning based on *Q-Learning* algorithm, which will be used to solve our problem and could be useful as a base to future developments.

Keywords: *reinforcement Learning*, *Q-Learning*, *framework*, *StarCraft*, real-time strategy games, maze.

Capítulo 1. Introducción

1.1 Motivación

Find a bug in a program, and fix it, and the program will work today. Show the program how to find and fix a bug, and the program will work forever.

Oliver G. Selfridge

En el pasado, instruir a un sistema informático para realizar una tarea requería definir un algoritmo para esa tarea y luego programarlo detalladamente. Esto conllevaba que tenía que haber un programador pendiente en todo momento para actualizar cualquier parte del código en el caso en que se encontrase un error o se introdujese una actualización [4], aparte de que se debía encontrar un algoritmo concreto por cada problema planteado.

En la actualidad, los sistemas informáticos pueden aprender a resolver un problema de forma autónoma a través de ejemplos de entrenamiento o por analogía a una tarea similar previamente resuelta [4]. También pueden mejorar significativamente su comportamiento o adquirir nuevas habilidades mediante prueba y error. Hay muchas maneras de llevar esto a cabo, y una de ellas es aprendizaje por refuerzo.

De entre todos los problemas que se pueden resolver mediante aprendizaje por refuerzo, hemos decidido enfocar nuestro trabajo en el mundo de los videojuegos, y emplear un juego de estrategia en tiempo real: *StarCraft*. De esta manera tenemos un dominio lo suficientemente complejo para realizar pruebas, pero a la vez bastante controlado. Asimismo nos hemos centrado en el movimiento de unidades en este juego, y en el problema concreto de hacer que una unidad aprenda a salir de un laberinto por sí sola.

Como nuestro objetivo es que el agente aprenda de forma autónoma a resolver el laberinto, queríamos evitar introducir mucho conocimiento del dominio de forma manual, dejando que el sistema aprenda sin necesidad de ello. Debido a esto, el proceso de aprendizaje es lento pues requiere realizar numerosas simulaciones.

Por otra parte, como los algoritmos de aprendizaje por refuerzo pueden resultar complejos, hemos desarrollado un *framework* reutilizable basado en el algoritmo *Q-Learning*, que facilita el uso de esta técnica en distintos dominios y nos permite centrarnos más en desarrollar las estrategias de aprendizaje para *StarCraft*. Este *framework*, asimismo, permitirá desarrollar proyectos basados en este algoritmo de una forma más rápida en el futuro. Además hemos podido probar distintas estrategias de recompensa, para valorar cuál es la más eficiente.

1.2 Objetivos

En el desarrollo de este trabajo, dada su complejidad, hemos perseguido los siguientes objetivos principales:

- **Estudio de técnicas de aprendizaje por refuerzo**, implementando aquellas que nos fueran más útiles o interesantes. Dado a que en las distintas asignaturas del grado no hemos tenido la oportunidad de estudiar estas técnicas de aprendizaje automático, queríamos desarrollar nuestro trabajo en torno a ellas.

- **Uso de videojuegos como dominio de pruebas**. Dado a que las IA de los juegos actuales no utilizan este tipo de técnicas, nos pareció interesante hacer un pequeño avance en la materia. Para desarrollar una IA debíamos demostrar que está fuera capaz de resolver problemas en dominios reales complejos y en tiempo real; por ello utilizamos el videojuego *StarCraft*. Este juego proporciona un dominio de aprendizaje perfecto, con una complejidad muy elevada propia de los RTS.

- **Desarrollo de un *framework* reutilizable basado en el algoritmo *Q-Learning***. Este *framework* debe proporcionar un entorno de desarrollo fácil y rápido para el uso de *Q-Learning* en distintos dominios. El diseño del mismo se basará en los patrones y directrices empleados en Ingeniería del Software.

Por último, y como objetivo transversal, queremos contribuir con nuestro trabajo a la comunidad de software libre, liberando el código y la documentación de todo nuestro proyecto.

1.3 Estructura del documento

La estructura de este documento se corresponde con las distintas fases que hemos seguido en nuestro proyecto.

El capítulo 3 explica las bases del aprendizaje automático y aprendizaje por refuerzo, centrándonos en el algoritmo *Q-Learning*, que es el que hemos implementado. En la segunda parte del capítulo describimos el videojuego *StarCraft*, que es el dominio sobre el que hemos realizado las pruebas. También hablamos acerca de la tecnología que permite crear los *bots* y controlar las unidades del juego.

El capítulo 4 describe la arquitectura general de nuestro proyecto y explica en profundidad el *framework* basado en *Q-Learning* que hemos desarrollado, llamado "*Teseo Q-Learning Framework*". En una segunda parte describimos las herramientas desarrolladas y/o empleado en el diseño de los experimentos y análisis de datos.

En el capítulo 5, debido a la complejidad de ejecutar multitud de pruebas en *StarCraft*, vamos a realizar experimentos en un dominio lógico desarrollado por nosotros. En estos experimentos vamos a calcular los parámetros del algoritmo más eficientes, además de hacer pruebas con distintas estrategias de recompensa.

En el capítulo 6 realizamos experimentos con los valores obtenidos en el capítulo anterior, pero esta vez en el dominio de *StarCraft*. Analizamos los resultados obtenidos y los comparamos con los resultados del dominio simplificado del capítulo anterior.

En el capítulo 7 y 8 explicamos las conclusiones finales de nuestro estudio y proponemos posibles líneas de trabajo futuro. En el capítulo 7 en castellano y en el 8 en inglés.

Por último, en el capítulo 9 explicamos las diferentes aportaciones individuales de los componentes del equipo al proyecto.

1.4 Organización del grupo de trabajo

Para el desarrollo de este proyecto hemos seguido una organización descentralizada, repartiendo la toma de decisiones a todos los componentes del grupo. Esto ha sido posible por el pequeño tamaño del grupo, y a la realización de reuniones semanales con el tutor. Durante estas reuniones exponíamos el trabajo realizado durante la semana y comentábamos los problemas encontrados, así como posibles soluciones para los mismos. Por último, planteábamos los objetivos para la próxima semana. Todas las tareas que se debían realizar fueron anotadas en la herramienta de gestión colaborativa *Trello* [5], donde íbamos organizando las tareas en: hechas, haciendo y pendientes de hacer.

Para el desarrollo del código hemos utilizado *GitHub* [2] como repositorio, empleando la herramienta de control de versiones *Git* [6]. Hemos mantenido una rama *master*, y diversas ramas de desarrollo para modificaciones. Cabe destacar que hemos basado el desarrollo en torno a pequeños prototipos de menor a mayor complejidad, pudiéndonos enfrentar a tecnologías que no conocíamos.

Para la documentación hemos usado *Google Docs* [7] y *Drive* [8] para poder compartir y editar de manera colaborativa.

Capítulo 2. Introduction

2.1 Motivation

In the past, to instruct a computer system to perform a task, it was required to define an algorithm for that task, and then program it in detail. This implied that there had to be always a programmer to update the code in case an error appeared or an upgrade was introduced [4], other than that it should find a specific algorithm for each problem posed.

Nowadays, computer systems can learn to solve a problems autonomously through training examples or by analogy to similar work previously determined [4]. They can also significantly improve their behavior or acquire new skills through trial and error. There are many ways to accomplish this, and one of them is reinforcement *Learning*.

Of all the problems that can be solved by reinforcement *learning*, we decided to focus our work in the video games world, and to employ real time strategy game: *StarCraft*. In that way, we have a complex test domain, but very controlled at the same time. We have also focused on the movement of game-units in this game, and on the specific problem of making a unit learn how to get out of a maze by itself.

As our main goal is the autonomously learning of the agent to solve the maze, we wanted to avoid introducing much domain knowledge manually, allowing the system to learn without it. Because of this, the learning process is slow as it requires performing numerous simulations.

Moreover, as algorithms of reinforcement learning can be complex, we developed a reusable *framework* based on the *Q-Learning* algorithm, which facilitates the use of this technique in different domains and allows us to focus more on developing learning strategies for *StarCraft*. This *framework* will also allow developing projects based on this

algorithm more quickly in the future. In addition we've been able to try different reward strategies to assess which is the most efficient.

2.2 Objectives

On the development of this work, due to its complexity, we have pursued the following main objectives:

- **Study of reinforcement *learning* techniques**, by implementing those that were the most useful or interesting. Based on the fact that in the different subjects of the Software Engineering Degree we didn't have the opportunity to study these techniques of automatic *Learning*, we wanted to develop our work around them.

- **Use of video games as a test domain**. Hence IA of the current games doesn't use this kind of techniques, we found it interesting to give a small advance in the field. To develop an AI we should demonstrate that it could solve real problems in complex domains and in real time; therefore we use the game *StarCraft*. This game provides a perfect learning domain, with its own very high complexity of the RTS.

- **Development of a reusable *framework* based on *Q-Learning* algorithm**. This *framework* should provide a quick and easy environment for using *Q-Learning* development in different domains. Its design will be based on the standards and guidelines used in software engineering.

Finally, and as a cross-cutting objective, we want to contribute with our work to the free software community, releasing the code and documentation of our entire project.

2.3 Document Structure

The structure of this document corresponds to the different phases we have followed in our project.

Chapter 3 analyze the basis of automatic learning and reinforcement *Learning*, focusing on the *Q-Learning* algorithm, which is the one we implemented. The second part of the chapter defines the *StarCraft* game, which is the domain where we performed the tests. We also talk about technology for creating bots and control units in the game.

Chapter 4 describes the general architecture of our project and points in detail the *framework* based on *Q-Learning* we have developed. In the second part we talk about the tools that we have developed and/or used in the design of experiments and data analysis.

Chapter 5, due to the complexity of implementing numerous tests in *StarCraft*, we will perform experiments in a logical domain developed by ourselves. In these experiments we will calculate the most efficient parameters of the algorithm, besides testing different reward strategies.

In chapter 6, we performed experiments with the values obtained in the previous chapters, but this time in the domain of *StarCraft*. In addition we analyze the results and compare them with the outcomes of the simplified domain of the previous chapter.

Finally, in Chapter 7 and 8 we explain the final conclusions of our study and propose possible lines for future work. Chapter 7 in Spanish and chapter 8 in English.

Chapter 9 explains the different individual contributions of the team components to the project.

2.4 Working Group Organization

To develop this project we have followed a decentralized organization, handing the decision to all members of the group, thanks to the small size of the group, and the conducting weekly meetings with the tutor. On these meetings we analyzed the week-work and we disclosed the problems found and achievable solutions. Finally, we settled down objectives for the following week. All tasks that should be performed were written down in the collaborative management tool *Trello* [5], where we were organized tasks as: made, doing and to do things.

To develop the code we used *GitHub* [2] as a repository, employing the tool Git version control with *Git* [6]. We have maintained a master branch, and various branches of development for amendments that arose. It is worth mentioning that we based the development around small prototypes of increasing complexity.

For documentation we used *Google Docs* [7] and *Google Drive* [8] to share and edit collaboratively.

Capítulo 3. Aprendizaje por refuerzo y *StarCraft*

3.1 Aprendizaje automático

El aprendizaje automático hace referencia a la cuestión de cómo construir programas que mejoren automáticamente mediante la experiencia [9]. Por ello es una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan aprender a los ordenadores. De manera más concreta, explora la construcción y estudio de algoritmos que puedan aprender y hacer predicciones en base a datos [9][10]. Los problemas de aprendizaje pueden ser clasificados en dos grandes bloques: aprendizaje supervisado y aprendizaje no supervisado.

Nosotros, en concreto hemos estudiado la rama de aprendizaje por refuerzo, la cual se engloba dentro del aprendizaje no supervisado.

3.1.1 Aprendizaje supervisado

En el aprendizaje supervisado el objetivo es predecir un valor estimado de salida basado en una serie de valores de entrada [11].

Finalmente se determina una función que establece una correspondencia entre las entradas y las salidas deseadas del sistema. De esta forma podemos llegar a predecir valores de salida para entradas no evaluadas. Dentro de este tipo de aprendizaje existen multitud de enfoques: regresión lineal y logística, árboles de decisión, redes neuronales, redes bayesianas, *SMVs*, *kNN*, *random forests*, etc. Algunos problemas comunes en los que se aplica este tipo de aprendizaje pueden ser los filtros de *spam*, el diagnóstico de enfermedades, el reconocimiento de escritura, o las predicciones de transacciones en bolsa.

3.1.2 Aprendizaje no supervisado

En el aprendizaje no supervisado no se intenta calcular un valor estimado de salida, si no que el objetivo es describir las asociaciones y patrones existentes entre un conjunto de valores de entrada [11].

Este aprendizaje se distingue por el hecho de que no hay un conocimiento *a priori*, ya que no partimos de ejemplos de entrenamiento etiquetados; no se tiene información sobre las categorías de esos ejemplos. Algunos problemas en los que se emplea son: problemas de *clustering* (agrupamiento), compresión de datos, sistemas de recomendación, etc. Dentro de esta categoría se encuentra el aprendizaje por refuerzo.

3.2 Aprendizaje por refuerzo

El aprendizaje por refuerzo comprende una serie de técnicas y problemas que surgen de la interacción entre un **Agente**, que intenta satisfacer un objetivo o meta, y su **Entorno** [12].

Por ello, cada vez que el agente realiza una acción en su entorno, el entrenador debe proporcionar una recompensa o una penalización para indicar la conveniencia del estado resultado. El objetivo de este agente es aprender, mediante estas recompensas, a elegir secuencias de acciones que produzcan estados favorables, y maximicen la recompensa acumulada. En la figura 1 podemos ver cómo el agente interactúa con su entorno.

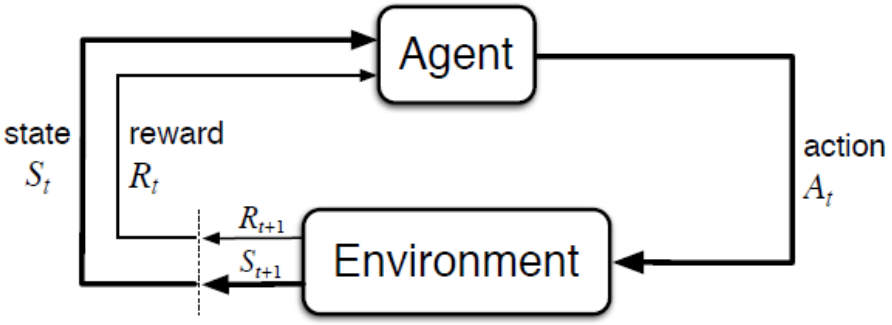


Figura 1. Funcionamiento del aprendizaje por refuerzo [13]

El agente existe en el entorno, descrito como un conjunto de posibles estados *S*. En cada paso t, el agente recibe una representación del estado del entorno S_t, y sobre esa base selecciona una acción A_t, de entre todas las posibles definidas en *A*. Esto hace que el estado actual del entorno se vea modificado S_{t+1}, por lo que el agente recibe una recompensa R_{t+1}, que puede ser positiva o negativa [13].

Según *Barto y Sutton* [13], más allá del agente y del entorno, se pueden identificar cuatro subelementos principales de un sistema de aprendizaje por refuerzo: una **política**, una **función de recompensa**, una **función de evaluación**, y, opcionalmente, un **modelo de entorno**.

La **política** define la forma de comportarse del agente en un momento determinado. En términos generales, la política es un mapeo de los estados posibles del entorno y las acciones que deben tomarse desde esos estados. La política es el núcleo del aprendizaje por refuerzo, en el sentido de que es la que determina el comportamiento.

Una **función recompensa** define el objetivo a alcanzar. Se asigna a cada estado percibido (o estado-acción) del entorno una recompensa, que indica la conveniencia del mismo. El objetivo es, como ya hemos mencionado, maximizar la cantidad total de recompensa acumulada. Además, la función de recompensa debe ser inalterable por el agente. La recompensa, sin embargo, puede servir como base para alterar la política. Por ejemplo, si una acción seleccionada por la política tiene una baja recompensa, entonces la política actuará para seleccionar alguna otra acción en esa situación en el futuro.

La mejor forma de entender esto es ver distintos ejemplos concretos: por ejemplo, para hacer que un robot aprenda a caminar los investigadores podrían proporcionar una recompensa en cada paso proporcional al avance. Si después deseáramos que este robot aprendiera a escapar de un laberinto, se le podría recompensar al salir del laberinto.

Mientras que una función de recompensa indica lo que es bueno en un sentido inmediato, la **función de evaluación** específica lo que es bueno a largo plazo. Se puede definir la

función de evaluación como la cantidad total de recompensa que el agente puede esperar acumular en el futuro a partir de ese estado; considerando las recompensas inmediatas y las recompensas disponibles en estados futuros. Por ejemplo, un estado podría obtener una recompensa inmediata baja, pero todavía podría tener un gran valor en la función de evaluación porque los estados sucesivos llevan a recompensas muy positivas. O a la inversa. El aprendizaje por refuerzo normalmente debe buscar acciones que provoquen funciones de evaluación de más valor, no de recompensa más alta, porque estas acciones obtienen una mayor cantidad de recompensa a largo plazo.

El cuarto elemento de algunos sistemas de aprendizaje por refuerzo es el **modelo del entorno**. Este es una forma de imitar el comportamiento del entorno. Por ejemplo, dado un estado y una acción, el modelo puede predecir el siguiente estado resultante y su recompensa. Este modelo se utiliza para hacer más eficiente la función de evaluación; ya que puede considerar estados o recompensas futuras, sin necesidad de que sean realmente experimentados.

3.3 *Q-Learning*

El algoritmo de *Q-Learning* (Watkins, 1989) intenta resolver el problema de cómo se debe actualizar la función de evaluación para que el agente pueda aprender cuál es la función óptima en un entorno no determinista, independientemente de la política a seguir.

Es complicado obtener la función óptima de forma directa porque la única información disponible es consecuencia de recompensas inmediatas. Como se puede ver, dada este tipo de información es más fácil generar una función de evaluación a partir de pares estado-acción. Este algoritmo aproxima iterativamente una función valor-acción inicial *Q(s,a),* representada en forma de tabla, hacia la función valor-acción óptima *Q*(s,a)* de la cual se extrae de forma directa la política óptima.

Para entenderlo mejor, podríamos plantearnos la siguiente pregunta: ¿cuál es la mejor acción que debe elegir el agente a cada paso? Una respuesta obvia sería elegir aquella acción que nos llevara a un estado con una función de recompensa mayor, por lo que se

maximizaría la suma de recompensa *r(s,a)*. Lamentablemente este modo de determinar la política óptima sólo funciona cuando el agente puede predecir el resultado inmediato (recompensa y estado sucesor) de cada posible acción, que en la mayor parte de los casos es imposible de determinar. ¿Entonces qué función de evaluación podemos utilizar cuando el agente se encuentra en un entorno arbitrario? La función Q nos da una solución.

3.3.1 La función Q

La función de evaluación *Q(s,a)* se define como el valor máximo de recompensa acumulada que puede ser obtenida desde un estado *s* tomando la acción *a*. En otras palabras, es igual a la recompensa inmediata *r* de ejecutar la acción *a* desde el estado *s*, más el resultado de seguir la política óptima desde entonces.

Dado que el conjunto de estados y acciones en la función *Q(s,a)* es discreto, podemos expresarlos en forma de tabla, en la que cada fila corresponde a un estado *s*, y cada columna al valor de la función resultante de escoger la acción *a*. En la figura 2 podemos observar un ejemplo de cómo se almacena esta información en una *QTabla*.

	Acciones (8)							
Estados (4)	1.0	1.0	0.98	0.88	0.0	0.2	0.75	1.0
	0.0	0.12	0.78	0.98	0.685	0.685	1.0	0.0
	1.0	1.0	1.0	0.557	0.0	0.712	1.0	0.0
	0.87	0.0	1.0	1.0	0.153	1.0	0.98	0.015

Figura 2. Ejemplo del almacenamiento de la *QTabla*

A la vista de la figura 2 podemos observar que el número de columnas corresponde al número de acciones posibles, y el número de filas corresponde con el número de estados. Aunque en este ejemplo mostramos sólo 4 estados, por lo general existen muchos más. En cada celda se expresa la probabilidad de realizar cada una de las acciones en cada estado posible.

Estas probabilidades son el resultado de la función de evaluación *Q*; en consecuencia, a esta tabla la denominaremos a partir de ahora *QTabla*.

Esta *QTabla Q(s,a)* se actualiza mediante la fórmula de *Q-Learning* (figura 3):

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$$

Figura 3. Fórmula del algoritmo *Q-Learning*

La fórmula de la figura 3 muestra precisamente la relación entre la recompensa obtenida y la recompensa acumulada:

a. *Q(s, a)* -> Indica el valor de la función *Q*, que se almacena en la *QTabla.*

b. *(1 - α)Q(s, a)* -> Valoración del aprendizaje ya almacenado en la *QTabla.*

c. *α(r + γmaxa'Q(s', a'))* -> Valoración del nuevo aprendizaje: la recompensa inmediata y la recompensa de seguir la política óptima.

3.3.2 Parámetros *Alpha* y *Gamma*

Para entender el funcionamiento de los parámetros es más fácil emplear una segunda expresión de la fórmula de *Q-Learning*, equivalente a la de la figura 3, empleada por *Sutton y Barton*:

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

Figura 4. Fórmula de *Q-Learning* empleada por Sutton y Barton [13]

En la expresión de la figura 4 se ve claramente que los valores *alpha* y *gamma* de la función son valores que afectan al aprendizaje, que valoran el conocimiento adquirido hasta el momento y la importancia sobre las recompensas futuras respectivamente. Ambos valores se encuentran en el rango [0,1]. A continuación los describiremos individualmente:

- ***Alpha***: es el ratio de aprendizaje; que como podemos ver en la figura 4, afecta a toda la segunda parte de la fórmula, que representa el aprendizaje adquirido; por ello valora la importancia de la recompensa recién adquirida, de forma que:

  - **Un valor cercano a 0**: actualizará la tabla muy lentamente, es decir, tiene más importancia lo que ya sabemos.

  - **Un valor cercano a 1**: actualizará la tabla más rápidamente, considerando sólo la información recién adquirida.

- ***Gamma***: es el factor de descuento; como vemos en la figura 4, este factor afecta únicamente a las posibles recompensas adquiridas en el futuro; por lo que determina la importancia de las recompensas futuras, de forma que:

  - **Un valor cercano a 0**: valorará más las recompensas inmediatas que el aprendizaje futuro.

  - **Un valor cercano a 1**: valorará más las recompensas futuras que las inmediatas.

Estos dos valores normalmente se obtienen de manera experimental; es necesario experimentar con distintos valores de estas constantes para encontrar los que funcionan mejor en cada tipo de problema.

3.3.3 Funcionamiento del algoritmo *Q-Learning*

El algoritmo *Q-Learning* se puede resumir en el siguiente pseudocódigo:

1. Inicializar la tabla de estados *Q(s,a)* para todo par estado *s*, acción *a*.
2. Mientras que no se haya alcanzado un estado de terminación (éxito o fracaso):

2.1. Observar el estado *s*.
2.2. Elegir una acción *a,* basada en los valores de la *QTabla*, y ejecutarla en el entorno.
2.3. Observar el siguiente estado ***s'*** y su recompensa *r*.
2.4. Aplicar la fórmula de *Q-Learning* actualizando el valor *Q(s,a)* de la *QTabla*, teniendo en cuenta el siguiente estado observado ***s'*** y la recompensa obtenida *r*.

Para entenderlo mejor, vamos a explicarlo paso a paso:

Lo primero es inicializar toda la tabla de estados. Esta se puede inicializar siguiendo varios criterios: con valores aleatorios, todos igual a 1, etc. Esto depende de qué política de selección de acciones vayamos a utilizar. En nuestro caso inicializamos todo a 1, para que todas las acciones tengan la misma probabilidad de ser elegidas.

Después se observa el estado actual en el que nos encontramos, y valoramos todas las posibles acciones desde dicho estado. De todas ellas se selecciona y ejecuta una acción concreta dependiendo de los valores almacenados en la *QTabla*.

Esta acción modificará el estado actual, y se obtendrá una recompensa en función del nuevo estado.

Con los datos obtenidos (estado anterior, acción ejecutada y recompensa) se calcula el valor de la fórmula de *Q-Learning*, y se actualiza en la *QTabla* la celda correspondiente al estado anterior y la acción que ha sido ejecutada. Con esta fórmula se aumentará o disminuirá el valor existente en la *QTabla* en función de si la recompensa ha sido positiva o negativa.

Una vez hemos actualizado la *QTabla*, y si nos hemos llegado a un estado final, volvemos a observar el estado actual en el que nos encontramos y seleccionamos una nueva acción.

El problema principal del algoritmo *Q-Learning* es encontrar una forma segura de aproximar el valor de *Q*, empleando únicamente una secuencia de recompensas inmediatas *r* que se propagan a lo largo del tiempo. Esto sólo se puede conseguir mediante numerosas iteraciones [9]. Es por ello que este algoritmo requiere de muchas ejecuciones para conseguir que el agente aprenda la política óptima *Q*(s, a)*, siguiendo un enfoque prueba-error. El número de repeticiones necesarias, asimismo, depende del número de estados y acciones posibles.

Se ha demostrado que *Q(s, a)* converge a *Q*(s, a)* si [9]:

1. El sistema se puede modelar como un MDP (*Markov Decision Process*) determinista.
2. Los refuerzos inmediatos están acotados.
3. Cada pareja *(s, a)* es visitada un número infinito de veces.
4. El entorno es estacionario (es decir, la probabilidad de transitar de *s* a *s'* ejecutando la acción *a* no varía en el tiempo).

3.3.4 Selección de acciones y exploración

El agente necesita una política para seleccionar las acciones a ejecutar en un determinado estado. Uno de los retos que se plantean en el aprendizaje por refuerzo, y no en otros tipos de aprendizaje, es el equilibrio entre *exploración* y *explotación*. Para obtener una gran recompensa, un agente de aprendizaje por refuerzo debe priorizar acciones que al intentarlas en el pasado le dieran un mayor beneficio, pero también tiene que explorar nuevas acciones y estados con el fin de descubrir estados que den mejores recompensas futuras [13]. El dilema es que se debe encontrar un buen equilibrio entre la *exploración* y la *explotación*. El agente debe tratar de ejecutar distintas acciones y favorecer progresivamente aquellas acciones que parezcan ser mejores.

Por ello se desarrollan mecanismos específicos de selección de acciones. En este sentido existen 3 políticas básicas: *greedy*, ***ε-greedy***, y *soft-max*.

- ➔ *greedy* elige la mejor acción, la que maximiza *Q(s, a).*
- ➔ ***ε-greedy*** elige la mejor acción con probabilidad ***ε***, en otro caso elige aleatoriamente.
- ➔ *soft-max* hace que todas las acciones sean equiprobables en un principio, y luego va modificando su comportamiento convirtiéndose en una política *greedy*, teniendo en cuenta un valor de aleatoriedad, llamado temperatura.

3.4 *StarCraft*

StarCraft es un juego creado por Blizzard Entertainment, y uno de los videojuegos de estrategia en tiempo real (RTS) más exitosos de la historia, teniendo muy buenas críticas entre los jugadores, junto con *Age of Empires II* (figura 4) [14].

Figura 4. *Age of Empires II*

Figura 5. StarCraft

Los RTS son un subgénero de los juegos de estrategia donde todos los sucesos y acciones suceden de forma continua, sin que haya ningún tipo de pausa. Son juegos en los que no hay turnos, y están pensados para jugar de forma dinámica y rápida. La misión es controlar a un ejército de personajes en tiempo real y sin pausa para la toma de decisiones. Las misiones generalmente consisten en avanzar y expandir al ejército a lo

largo de un territorio, tomando las bases enemigas y a la vez construyendo bases para incrementar los recursos disponibles. Estos recursos son cosechados mediante el control de ciertos tipos de unidades dedicadas a este fin. Más específicamente, los típicos juegos RTS cuentan con recolección de recursos, construcción de bases, desarrollo tecnológico, y el control indirecto de unidades [15].

Este tipo de juegos, además suelen tener un gran componente estratégico a dos niveles: un nivel denominado "*macro*" donde se deben tomar decisiones en torno a la economía, expansión y producción de recursos; y un nivel "*micro*" en el que se deben tomar decisiones tácticas en cuanto al control del ejército [16].

StarCraft (figura 5) se encuentra ambientado en una historia futurista de ciencia ficción con gráficos 2D en perspectiva isométrica, en la que tres razas provenientes de distintos planetas, los *Terran*, *Zerg* y *Protoss*, luchan por la supremacía y la supervivencia. Cada raza tiene acceso a unidades únicas, las cuales desempeñan papeles concretos en el campo de batalla. Combinar unidades distintas para formar ejércitos versátiles es un modo de obtener la victoria [17].

Los *Terran* (figuras 6 y 7) son la raza humana, exiliados, expulsados de la tierra emprenden su viaje a través de las estrellas sobreviviendo en nuevos planetas a duras penas.

Figura 6. Logo de la Raza *Terran*

Figura 7. Ejemplo de juego con la Raza *Terran*

Los *Zerg* (figuras 8 y 9) son insectos altamente evolucionados, capaces de adaptarse a las condiciones más adversas. Viajan de planeta en planeta arrasando todo a su paso, con una mente colmena y se llaman a sí mismos el enjambre.

Figura 8. Logo de la raza *Zerg*

Figura 9. Ejemplo de juego con la raza *Zerg*

Los *Protoss* (figuras 10 y 11) son una poderosa civilización alienígena altamente avanzada tecnológicamente, fanáticos en sus creencias y en su supremacía exterminan planetas enteros en un intento por acabar con los *Zerg*, destruyendo a su vez a los *Terran*.

Figura 10. Logo de la raza *Protoss*

Figura 11. Ejemplo de juego con la raza *Protoss*

Posteriormente se puso a la venta su única expansión oficial, *StarCraft*: *Brood War*. Además, el fabricante facilita un servidor gratuito, Battle.net, para que los jugadores puedan competir en el modo multijugador a través de Internet. En las partidas multijugador se obtiene la victoria cuando se destruyen todos los edificios del enemigo, o si los demás jugadores se rinden.

El juego ha recibido varias actualizaciones en forma de parches, la versión más actual (y sobre la que hemos ejecutado nuestra aplicación) es la 1.16.1. Las versiones anteriores han ido puliendo el juego haciendo que las razas fueran más equilibradas, eliminando errores de programación, mejorando inteligencia artificial (IA), y añadiendo nuevos elementos de juego.

3.4.1 *BWAPI*

Brood War Application Programming Interface (***BWAPI***) es un *framework* libre de C++ usado para la creación de módulos de IA para *StarCraft*: *Brood War*. Utilizando *BWAPI*, los programadores pueden conseguir información sobre el estado del juego en *StarCraft* (jugadores, unidades, edificios…), a la vez que poseen una gran variedad de comandos para dirigir las unidades, pudiendo crear una IA personalizada [18].

Figura 12. Interfaz gráfica de *BWAPI*

En la figura 12 se puede ver el juego ejecutándose con el módulo de *BWAPI*. Este muestra de manera predeterminada las posiciones de las unidades del jugador.

Por defecto, *BWAPI* sólo revela las partes visibles del estado del juego a los módulos de IA. Pero no tiene acceso a la información de las unidades que no son visibles por el jugador en ese momento. Esto permite a los programadores escribir IA competitivas sin hacer trampa ya que deben planificar y operar bajo las condiciones de información normales, como todos los demás jugadores. Además, de forma predeterminada, *BWAPI* desactiva la interacción del usuario, relegando al usuario a ser un espectador mientras el módulo de IA se esté ejecutando. Esto se hace para asegurar que el ganador lo hace exclusivamente basado en la programación y algoritmos de IA en sí mismos, más que por la intervención humana.

Actualmente existen dos librerías que proporcionan una interfaz java al *framework BWAPI*. Estas son JNI-*BWAPI* [19] y *BWMirror*. Al realizar nuestra aplicación en java inicialmente íbamos a utilizar el API de JNI-*BWAPI*, pero tras investigar un poco más descubrimos que *BWMirror*, una API basada fuertemente en el JNI, tenía una interfaz mucho más amigable y estaba diseñado específicamente para estudiantes interesados en desarrollar IA de *StarCraft*. Es una aplicación de código libre [20] que es utilizada en distintas competiciones a nivel internacional, como la de la universidad de Bratislava [21].

3.4.2 Competiciones

Con la aparición de *BWAPI* se facilita el acceso al control del *StarCraft* a nivel de código de manera gratuita. Gracias a esto, una gran cantidad de público se ha puesto a hacer sus propios programas. En concreto, se han puesto a hacer sus propias IA capaces de ganar a la máquina del juego, Generando una gran competitividad y por ello se han generado una serie de competiciones a nivel internacional. Dichas competiciones, a las cuales acuden estudiantes y desarrolladores de IA de todas partes del mundo, tienen

cierta visibilidad y consiguen una gran participación, cada vez mayor, y un elevado nivel de competitividad.

El propósito de este tipo de concursos es fomentar y evaluar los avances de la investigación en IA aplicada a los juegos del género RTS. Estos juegos en tiempo real suponen un reto mucho mayor para la investigación en IA que el ajedrez debido a la información oculta y los vastos espacios de estados y posibles acciones. Además de la necesidad de actuar con la mayor rapidez posible.

Entre estos torneos se pueden destacar los siguientes.

- *AIIDE StarCraft AI Competition; asociado a la conferencia americana de inteligencia artificial y entretenimiento digital interactivo* [22].
- *IEEE CIG 2014 StarCraft AI Competition; asociados a la conferencia internacional de inteligencia computacional y juegos* [23].
- *Student StarCraft AI Tournament* (SSCAIT); una competición con fines educativos que pretende motivar a estudiantes de informática a usar distintas técnicas de inteligencia artificial [24] .

Los juegos de estrategia en tiempo real han supuesto un nuevo dominio para los investigadores de inteligencia artificial, los cuales usan estos dominios como *'testbeds'* (bancos de pruebas). Estos *testbeds* son altamente complejos y requieren dividir su dominio en distintos subproblemas para un manejo más óptimo [25].

En las actas de la competición asociada al congreso AIIDE, encontramos numerosos trabajos que combinan aprendizaje automático con otras técnicas de IA en juegos de estrategia en tiempo real. La mayoría de estos trabajos se centran en la parte del combate entre unidades. Esta tarea implica una carga muy grande de trabajo para el aprendizaje automático, ya que existe un gran volumen de estados/acciones. Este volumen es tan grande que normalmente se necesita la modificación o combinación de varios algoritmos de aprendizaje [20] porque las unidades pueden tratarse de distintas formas (en grupo, de forma independiente, etc.) [26]. En algunos trabajos se abstrae esta

parte para no tratar con las unidades en concreto, siendo únicamente necesario establecer la estrategia a seguir [27].

Capítulo 4. Arquitectura, *Teseo Q-Learning Framework*, y herramientas adicionales

En este capítulo vamos a explicar cuál es la arquitectura general de nuestro proyecto, la arquitectura del *framework* reutilizable de *Q-Learning* que hemos desarrollado (*Teseo Q-Learning Framework*), y otras herramientas auxiliares que hemos utilizado para generar las pruebas, analizar y visualizar los resultados obtenidos.

4.1 Arquitectura del sistema

La figura 13 muestra cómo se conectan los distintos módulos que componen nuestro sistema:

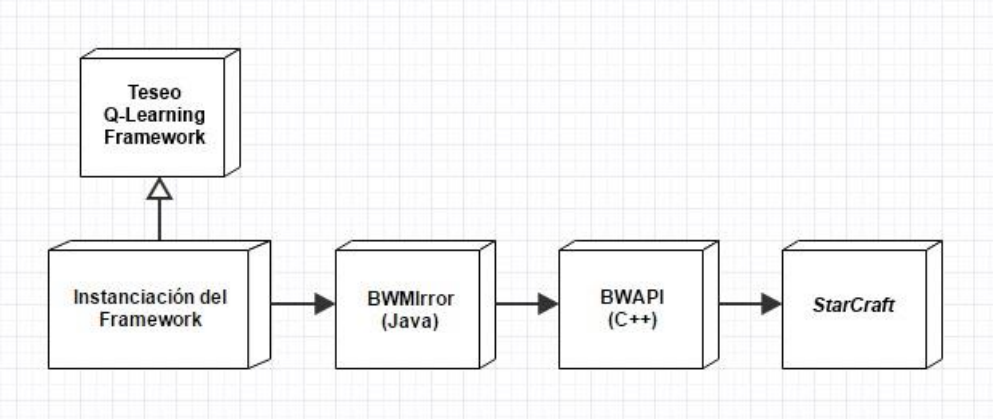


Figura 13. Arquitectura del sistema

Para empezar hemos desarrollado un *framework* reutilizable (***Teseo Q-Learning Framework***) para el aprendizaje mediante *Q-Learning*. Este *framework* contiene las clases y métodos básicos para el funcionamiento del algoritmo bajo cualquier dominio.

En este caso, vamos a instanciarlo sobre *StarCraft*, definiendo métodos, estados, acciones concretos del mismo.

Para que pueda funcionar sobre *StarCraft*, nuestra **instanciación del *framework*** hace uso de las clases y métodos ofrecidos por la librería *BWMirror*, que "es una API en Java que sirve para encapsular dentro de objetos Java todas las clases, constantes y enumerados que ofrece *BWAPI*, mientras que provee de una interfaz idéntica a la que ofrece la original *BWAPI* en C++" [28].

La API ***BWMirror*** traduce los mensajes y se comunica con *BWAPI* (escrita en C++) mediante JNI (*Java Native Interface*), y por último es la API de *BWAPI* la que interactúa directamente con el videojuego en tiempo real. ***BWAPI*** es capaz de leer el estado de ***StarCraft*** y realizar modificaciones (mandar comandos) en el mismo, que se ven reflejados en la pantalla. Es importante resaltar que en ningún momento tenemos acceso al código fuente del juego.

Hay que tener en cuenta que *BWAPI* está desarrollado para entornos que dispongan de un bucle de juego guiado por eventos. Esto se concreta en que tiene 3 métodos principales: *"onStart()",* que se ejecuta una sola vez al comenzar la partida; *"onEnd()",* que se ejecuta al acabar la partida; y *"onFrame()"* que se ejecute a cada instante durante la partida. Dentro de este último evento se debe localizar toda la lógica de IA, en la que el *bot* decide qué hacer.

4.2 *Teseo Q-Learning Framework*

En un principio, empezamos a desarrollar nuestra aplicación incluyendo toda la mecánica de aprendizaje dentro de una única clase java, que se comunicaba directamente con *BWMirror*. Dentro de esta clase incluimos métodos encargados de todo el proceso de aprendizaje (que gestionaban recompensas, actualizaban la *QTabla*, y modificaban el entorno). Esto hizo que cuanto más crecía en complejidad, más costoso era su mantenimiento.

En este punto teníamos dos frentes abiertos: *Q-Learning* y *StarCraft*, y afrontarlos a la vez era demasiado costoso. Por ello decidimos centrarnos al principio sólo en la parte de aprendizaje, trabajando con un dominio simplificado (que se explica en detalle en el capítulo 5). A esto se sumaba que cuando quisiéramos trasladar toda esta lógica de aprendizaje a un laberinto en *StarCraft*, iba a ser un proceso muy complejo de refactorización y arreglo de dependencias.

Teseo Q-Learning Framework surge de la necesidad generalizar este código enfocado al aprendizaje, de manera que cumpliera con los principios de facilidad de uso, flexibilidad, y portabilidad. Se planteó como un conjunto de clases e interfaces que constituyeran un *framework* reutilizable, que pudiéramos instanciar tanto en nuestro laberinto lógico como en un laberinto en el juego real, con el mínimo coste. En el siguiente apartado desarrollaremos el diseño y arquitectura de este *framework*.

4.2.1 Diseño y arquitectura

Para el diseño del *Teseo Q-Learning Framework* fue imprescindible consultar documentación acerca de los principios generales de *Q-Learning*.

En la siguiente imagen (figura 14) se puede ver un esquema con las principales interacciones del algoritmo:

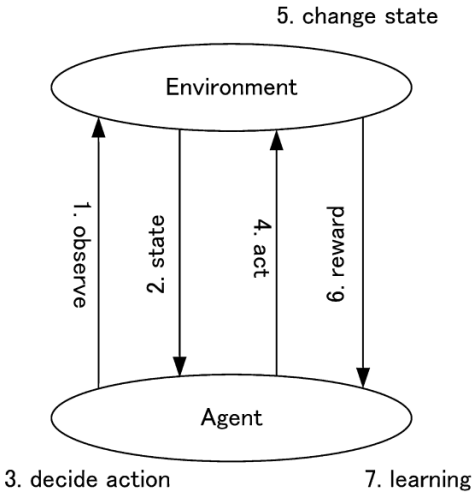


Figura 14. Interacciones del algoritmo Q-Learning [29]

Según el esquema, los principales actores son el **agente** (el jugador) y el **entorno** (nuestro mapa). Estos dos actores se comunican entre sí mediante estados, acciones y recompensas. Por eso, estos son los tres elementos principales. La arquitectura final de nuestro *framework* se puede explicar mediante los siguientes diagramas de clases:

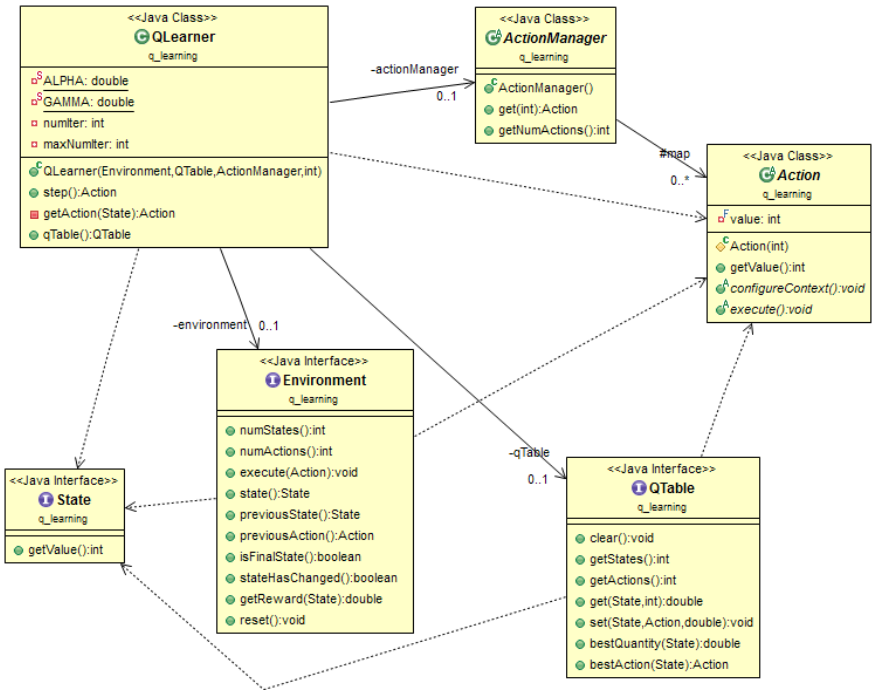


Figura 15. Diagrama de clases de nuestro *framework*

Como se puede apreciar en el diagrama de la figura 15, nuestro *framework* se compone principalmente de 3 clases y 3 interfaces:

- **interface *QTabla***: esta interfaz provee métodos de acceso a la estructura en la que se almacene la *QTabla*. Es una interfaz porque de esta manera es independiente a la implementación concreta que se le quiera dar a la estructura de datos. Los métodos principales de la interfaz *QTabla* son:

- *public double get(State state, int a);* que obtiene el valor de la *QTabla* para un par estado/acción determinados.

- *public void set(State state, Action action, double quantity);* que actualiza el valor de la *QTabla*.

- **class *QTabla*_Array**: como se ve en la figura 16, es una implementación de *QTabla* mediante una matriz de *doubles*. Mediante esta estructura de datos se consigue O(1) en las operaciones *get* y *set*. En cambio el coste de obtener la acción mejor valorada es de O(n), siendo n el número de acciones posibles para un determinado estado.

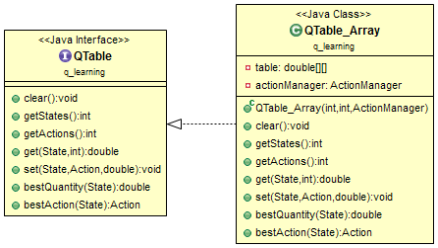


Figura 16. Interface *QTabla*

Dependiendo del número de estados y acciones de nuestro dominio nos puede interesar utilizar esta clase directamente o implementar la interfaz de otro modo, de manera más o menos compleja o eficiente. Por ejemplo, una implementación basada en tablas *Hash*, no tendría que representar las posiciones de estados no visitados; de este modo ganaría en memoria pero perdería rendimiento.

- ***abstract class Action***: esta clase (junto a la clase *ActionManager*) compone una variante del patrón *"Command"* en el que la propia acción es responsable de ejecutarse sobre el entorno. De esta forma conseguimos separar al módulo de aprendizaje de la lógica de ejecución de las acciones y los cambios que estas tienen en el entorno. Mediante este patrón conseguimos que sea muy sencillo añadir nuevas acciones, simplemente creando clases que hereden de esta primera. Además podemos encapsular el funcionamiento de cada acción, pudiéndose invocar de manera uniforme.

Por tanto está definida como una clase abstracta, de la cual deberán heredar todas las posibles acciones de nuestro dominio. Cada acción tendrá un identificador (entero en este caso) que servirá como índice en la *QTabla*(*state, action*). En cada una de las clases hijas habría que implementar las siguientes funciones:

- *public abstract void configureContext();* que debe facilitar a la acción las referencias de los objetos sobre los que se ejecutará. Ya que en nuestro dominio el comando se ejecutará sobre elementos y unidades de *StarCraft*, es necesario que disponga de ellos.

- *public abstract void execute();* que define cuál es el comportamiento de la acción. Las acciones en nuestro dominio corresponden a los movimientos en las 8 direcciones (N, NE, E, ...).

- ***abstract class ActionManager***: esta clase conforma el gestor de órdenes de nuestro patrón *"Command"*. Contiene un *HashMap<Integer, Action>* en el que se deben registrar las distintas acciones con su correspondiente identificador. Contiene métodos para obtener la acción asociada a cada ID numérico, y para saber el número total de acciones que el agente puede ejecutar.

- ***interface State***: esta interfaz representa el estado actual del juego a nivel lógico, y contiene un sólo método, *getValue(),* que se utilizará para indexar la *QTabla*. Cada estado debe saber cómo traducirse a un identificador único. En el caso de *StarCraft* las variables que definen el estado podrían ser el tiempo transcurrido, número de unidades, posición de cada unidad, dimensiones del tablero, nivel de vida de las unidades, etc. Debe ser el programador el que decida qué elementos debe tener en cuenta a la hora de codificar los estados en cada dominio.

- ***interface Environment***: esta es posiblemente la interfaz más compleja de implementar del *framework*, ya que debe contener métodos concretos que faciliten a la clase *QLearner* la información necesaria acerca del dominio sobre el cual se está ejecutando. Los métodos más importantes de esta interfaz son:

- *public void execute(Action action)*; ejecuta una acción en el estado actual. Deberá llamar a *configureContext()* de la acción y luego a *execute()*.
- *public State state();* devuelve el estado actual.
- *public State previousState();* devuelve el estado anterior.
- *public Acion previousAction();* devuelve la última acción tomada.
- *public boolean stateHasChanged();* comprueba si ha cambiado el estado desde el estado anterior.
- *public double getReward(State state);* devuelve la recompensa que se le da al agente en un determinado estado.

- ***class QLearner***: es la clase principal del *Teseo Q-Learning Framework*, que integra todas las anteriores. Esta clase se encarga de ir eligiendo acciones a cada paso y actualizando los valores de la *QTabla* en función de las recompensas que vaya obteniendo. Los 2 principales métodos de esta clase son:

  - *protected Action getAction(State state);* que elige una acción a ejecutar en función del estado actual. Esta acción se puede elegir mediante distintas políticas: aleatoria, probabilística, de máximos, *greedy*, ***ε****-greedy*, *soft-max*… En nuestro caso hemos decidido elegir una política probabilística, esto se concreta en que sumamos todos los valores de probabilidad de la *QTabla* para ese estado, y buscamos un número aleatorio entre 0 y el siguiente sumatorio:

$$\sum_{i=0}^{num\_actions} QTabla(\text{estado})(\text{i})$$

  De esta forma, las acciones con una mayor valor en la *QTabla* tendrán más probabilidad de escogerse (e irán reforzándose/debilitándose en función del aprendizaje recibido). En cambio, las acciones que hayan llegado a un valor 0 en la *QTabla*, no se podrán elegir nunca más. Esta política nos ofrece una buen equilibrio entre exploración de nuevos caminos y explotación de caminos considerados positivos.

- *public Action step()*; método en el que se elige y ejecuta una acción en base al estado actual. Con la recompensa obtenida se actualizan los valores de la *QTabla*. Dado a que las acciones tardan cierto tiempo en ejecutarse y la recompensa no se conoce hasta que la acción termina, nuestro método *step()* comienza desde el paso de "recompensa" del algoritmo (ya explicado en la sección 3.2 de este documento). Posteriormente actualizamos la *QTabla*, y elegimos y ejecutamos una nueva acción. Podemos ver en el siguiente diagrama de secuencia (figura 17) cuál es la interacción entre las distintas clases e interfaces.

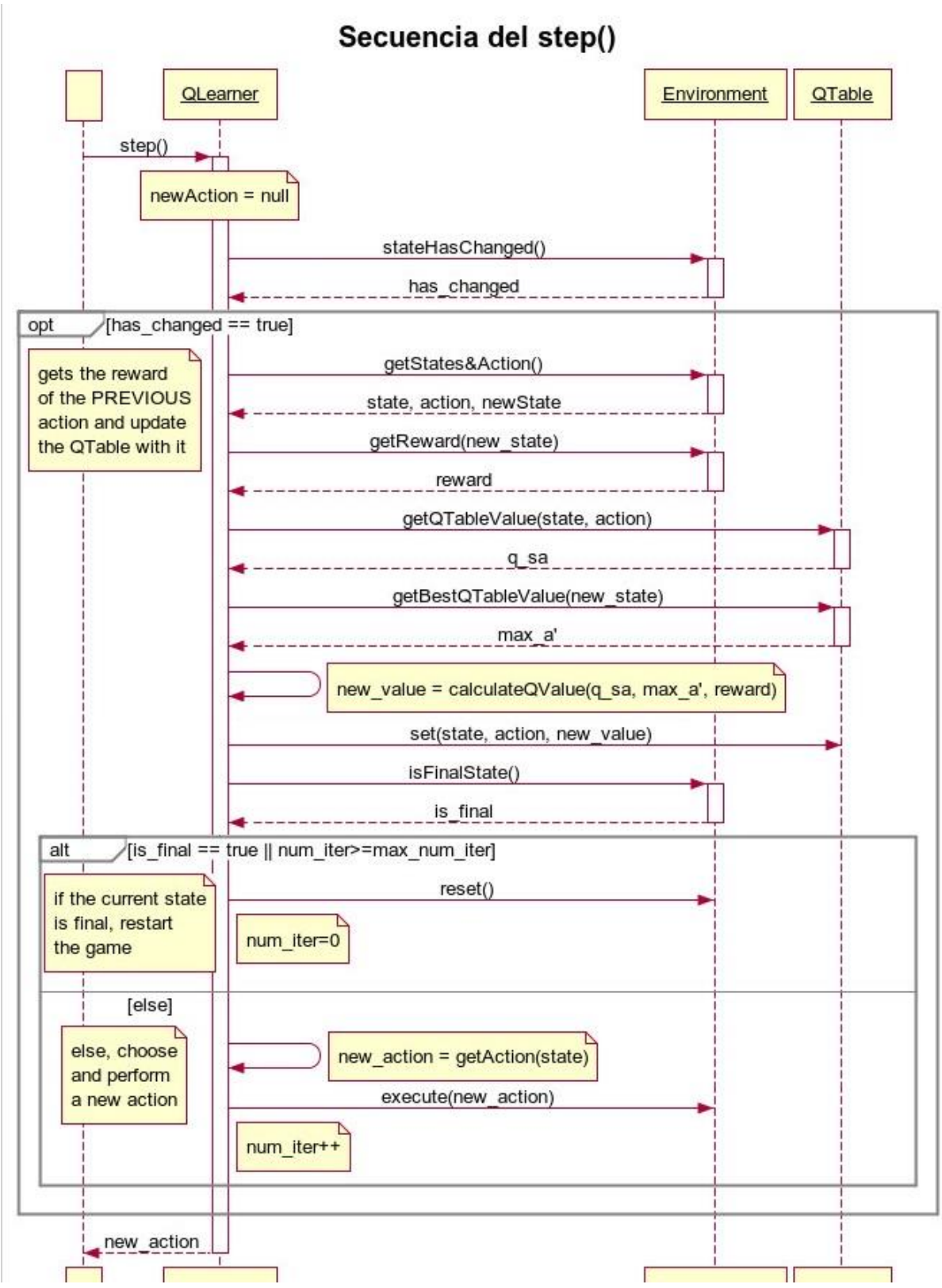


Figura 17. Diagrama de secuencia del *step()*

En el diagrama de la figura 18 podemos ver la secuencia para elegir la acción *getAction(state)*:

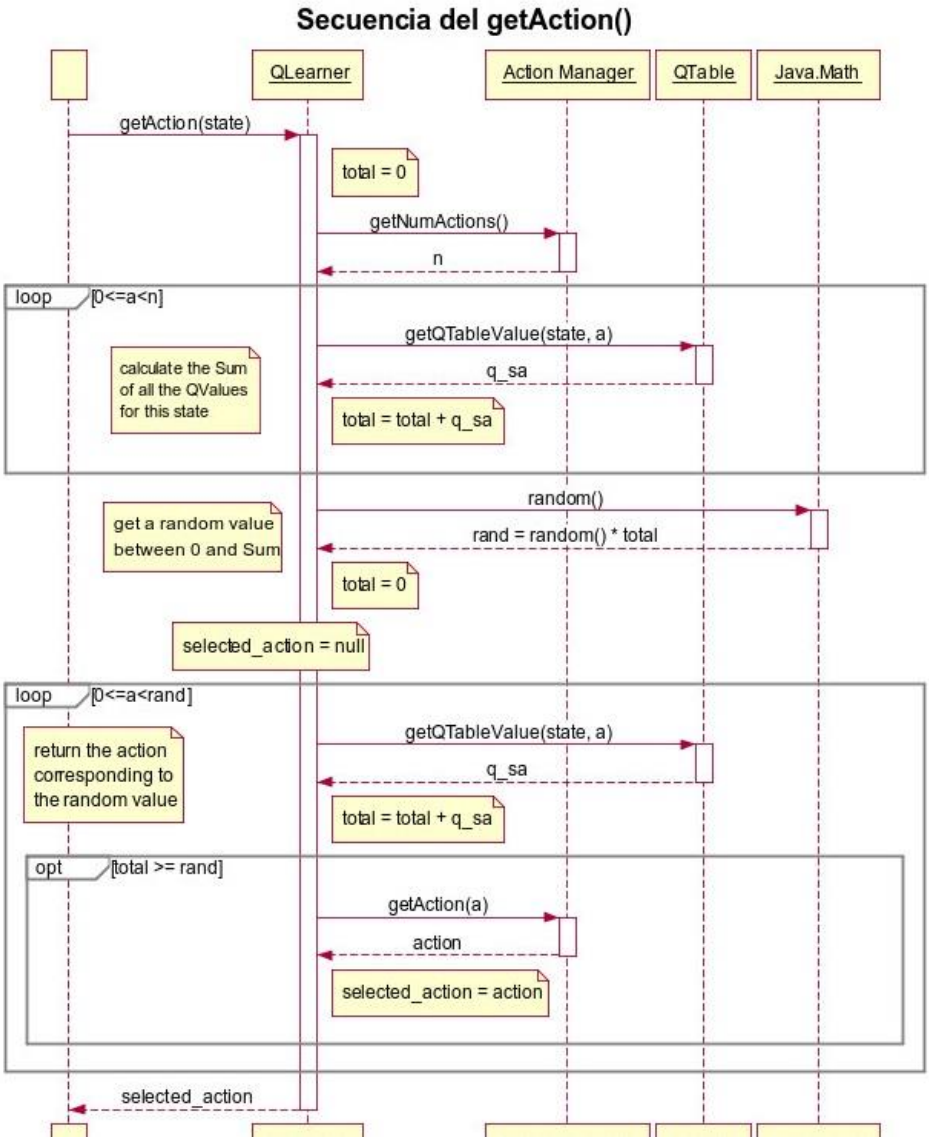


Figura 18. Diagrama de secuencia del *getAction()*

Con todo esto hemos logrado crear un diseño reutilizable, que puede ser instanciado en diversos dominios, simplemente implementando las interfaces necesarias y heredando de las clases propuestas. Dado el nivel de abstracción aplicado se podría aplicar sobre todo tipo de dominios de aprendizaje, incluso sobre dominios físicos (robótica), aunque

con la complejidad de codificar los estados y acciones que un dominio real conlleva. Todo esto depende de la instanciación concreta que se haga sobre el *framework*.

En el siguiente apartado podemos ver un ejemplo de instanciación del *Teseo Q-Learning Framework*, en este caso sobre *StarCraft*.

4.2.2 Ejemplo de instanciación

En este apartado vamos a explicar cómo sería la instanciación del *framework* en el dominio de *StarCraft*.

Lo primero será implementar el *interface State*; (figura 19) y ya que sólo vamos a movernos por un laberinto, únicamente nos interesa la posición de nuestra unidad. Para codificar el estado de manera única lo haremos empleando un estado por casilla, y mediante el cálculo de *value = positionY * board_width + positionX*.

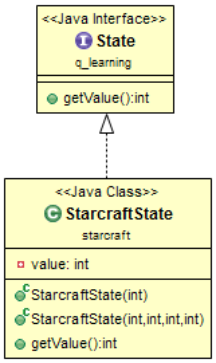


Figura 19. Instanciación de la I*nterface State*

Después debemos implementar las distintas acciones. El diagrama general lo podemos ver en la figura 20. Para ello hemos creado una nueva clase abstracta *StarCraftAction* que hereda de *Action*. En esta clase abstracta básicamente implementamos el método *configureContext()*, ya que el contexto es el mismo para todas las acciones que vamos a definir.

Una vez tenemos esta clase, podemos ir heredando de *StarCraftAction* para ir creando las distintas acciones, en nuestro caso: *MoveDown*, *MoveDownLeft*, *MoveLeft*, etc. En

cada una de estas acciones implementamos la función *execute()* obteniendo la posición del jugador, y ordenándole que avance a una nueva posición (en función de cuál sea el comando).

Para que estos comandos funcionen debemos registrarlos en el constructor del *StarCraftActionManager*.

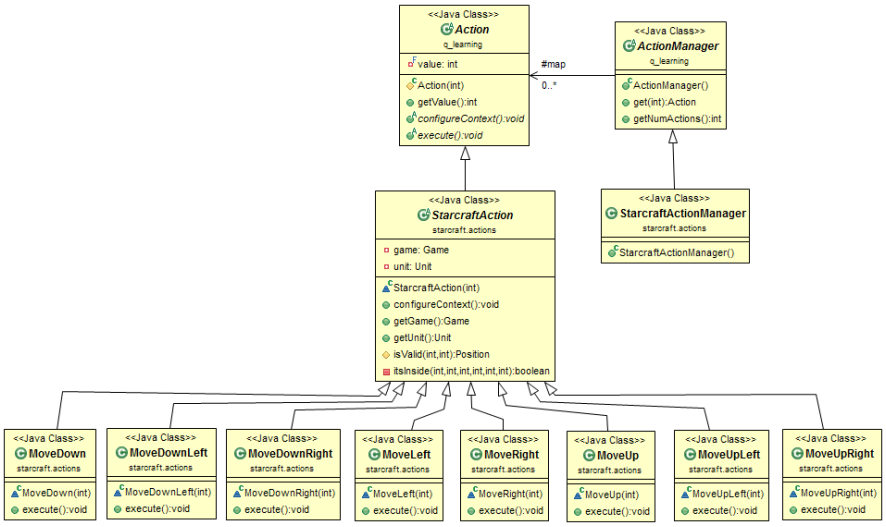


Figura 20. Instanciación de la clase *Action* y *Action Manager*

Por último deberemos realizar la implementación de *Environment*, que es la más compleja. En ella tenemos que implementar cada una de las funciones de la interfaz, con datos concretos correspondientes al dominio *StarCraft*. Por ejemplo, la función *stateHasChanged()* la implementamos como *return !unit.isMoving();* ya que consideramos que la unidad ha cambiado de estado cada vez que deja de moverse.

También almacena todos los datos necesarios para el funcionamiento de los distintos métodos:

- Recompensas de ganar y perder.
- Número máximo de pasos que permitiremos dar a la unidad en un experimento.

- Estrategia de recompensa usada para resolver el problema.
- Estado del juego.
- Unidad que deberá salir del laberinto.
- Estado y acción del paso anterior. Estas se usan para actualizar la *QTabla*.
- Estado inicial de la unidad y estado final (meta). Se utilizan para reiniciar el experimento.

El método más interesante de esta clase es *public double getReward(State state)*, al ser el que decide qué recompensa concreta corresponde a cada estado. Ya que vamos a dar la posibilidad al usuario de elegir distintas estrategias de recompensa, en este caso hemos creado un enumerado con las distintas estrategias, y en función de la que se le pase en la instanciación podemos dar una recompensa u otra. La estrategia elegida tendrá un gran impacto en el algoritmo de aprendizaje. Un ejemplo de estrategia de recompensa sería:

```
private double basicStrategy() {
      double reward = this.default_reward; //reward = 0.0
      if(hasLost()) { //if the unit doesn't exist (lost game)
            reward = this.lost_reward; //reward = -1
      } else if(hasWon()) { //if the unit reaches the goal
            reward = this.won_reward; //reward = 1000.0
      }
      return reward;
}
```

Por último, falta crear la clase principal que haga ejecutar todo este código. El método principal de esta clase se encarga de instanciar el resto de clases (el entorno *StarCraft*, unidades de *StarCraft*, posiciones de salida y meta, *QTabla* con estados y acciones, y el QLearner). Una vez instanciadas las clases, y conservando la referencia del *QLearner*,

únicamente nos queda llamar al método *step()* del QLearner en cada *frame* (dentro del bucle de juego).

Por último, se debe tener en cuenta que este *framework* tiene en cuenta la lógica orientada a eventos propia de los videojuegos, mediante un bucle de juego. Este bucle "infinito" de juego funciona de forma automática en *StarCraft*, pero en otras aplicaciones (como nuestro laberinto lógico) hay que simularlo mediante un bucle infinito controlado por el programador.

4.3 Herramientas de diseño de los experimentos y análisis de datos

Para realizar los distintos test que se explicarán en los capítulos sucesivos se han empleado algunas herramientas auxiliares para facilitar y automatizar todo el proceso. En este apartado vamos a explicar brevemente algunas de ellas.

- **Generador de laberintos lógicos**: como hemos decidido realizar las pruebas de aprendizaje sobre un laberinto lógico, por ello hemos desarrollado una herramienta con interfaz gráfica que nos facilita el proceso de crear los mapas lógicos. Se explicará con más detalle el funcionamiento de este tipo de laberinto lógico en el capítulo 5.

Figura 21. Interfaz gráfica del generador de laberintos lógicos

En la herramienta podemos seleccionar de forma interactiva el tamaño del mapa, paredes del laberinto, inicio y final, e incluso añadir trampas. Una vez le damos al botón “SALVAR LABERINTO” nos genera un fichero de texto plano en el que cada elemento está codificado mediante un número (0 = vacío, 1 = pared, 2 = inicio, 3 = meta, 4 = obstáculo). En la imagen de la figura 21 podemos ver un ejemplo de la herramienta:

La herramienta ha sido desarrollada en lenguaje Java y empleando la librería gráfica *Swing*. Además se ha desarrollado reutilizando el código de una práctica que tuvimos que hacer para la asignatura de Ingeniería del Conocimiento. Esta práctica consistía en aplicar el algoritmo *A** para resolver un laberinto, en un mapa de dos dimensiones. Como requisitos, la aplicación tenía que ser capaz de cargar un mapa, con un punto de partida, otro de llegada, y paredes que no se podían sobrepasar (los mismos requisitos que tiene nuestro laberinto lógico). Partiendo de esta base, decidimos implementar la carga y salvado de mapas, que realizamos mediante un *parser* a texto plano, y ampliarlo con la posibilidad de añadir trampas.

- **Librería para exportar a Excel**: Ya que hemos generado multitud de *logs* con los resultados de los test, hemos empleado una librería en java que exporta los datos en hojas de cálculo Excel, con las que luego generamos las gráficas automáticamente (desde código). Para ello hemos utilizado la librería *Apache POI* [30], capaz de escribir y leer ficheros tanto en Excel como en Word.

- **Editor de mapas de *StarCraft***: para generar los mapas en *StarCraft* hemos utilizado la herramienta oficial "Editor de Campañas de *StarCraft*", que se distribuye con el juego. Esta herramienta nos permite crear todo tipo de mapas pudiendo añadir unidades, razas, modificar número de jugadores, etc. La figura 22 es una captura de pantalla de la misma.

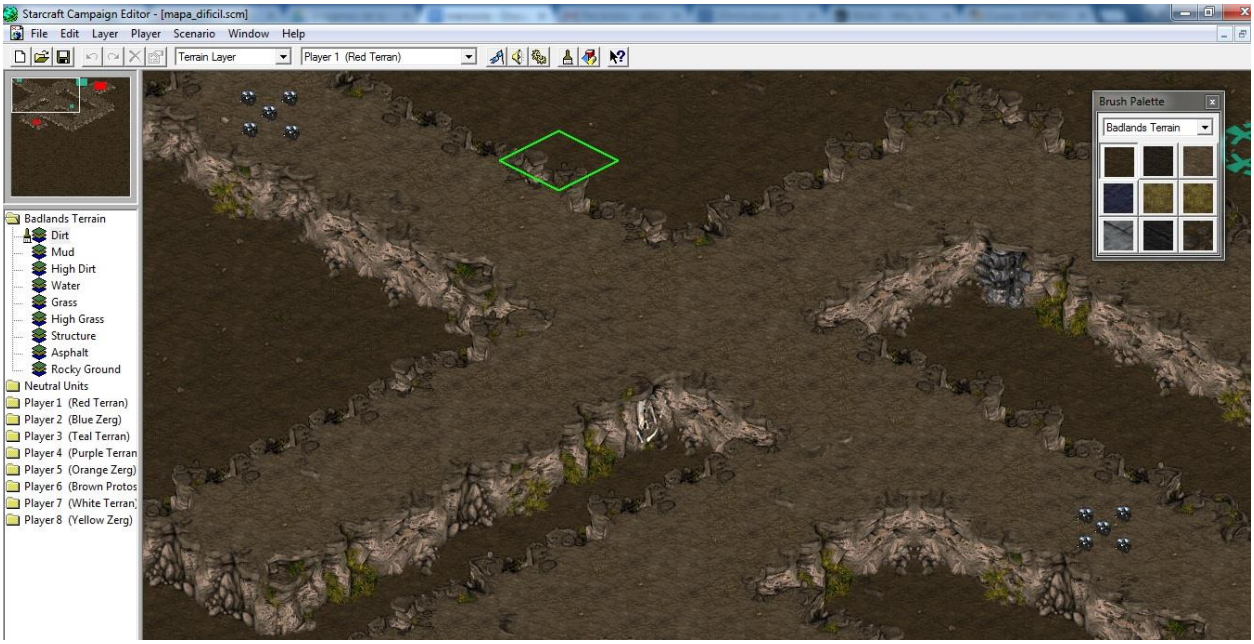


Figura 22. Editor de mapas de *StarCraft*

Capítulo 5. Resolución de laberintos lógicos

En un principio empezamos a trabajar realizando las pruebas de aprendizaje sobre *StarCraft*, pero estas pruebas eran demasiado costosas debido a que el juego se tiene que ejecutar casi en tiempo real, tardando casi un día en realizar un experimento completo. Esto hizo que fuera inasumible realizar gran cantidad de pruebas en este dominio, y por eso decidimos realizar estas pruebas dentro de un dominio simplificado, que hemos denominado “laberinto lógico”. Además, de esta forma podíamos centrarnos en el estudio del algoritmo *Q-Learning*, sin tener que lidiar con problemas procedentes de *BWAPI* o *StarCraft*. El laberinto lógico nos permitía ejecutar los experimentos mucho más rápido y de esta forma pudimos realizar multitud de experimentos para elegir los parámetros óptimos del algoritmo o las estrategias de recompensa más favorables, que trasladamos luego a *StarCraft*.

En este capítulo mostraremos cómo funciona el laberinto lógico, así como las pruebas realizadas y resultados obtenidos con los tres elementos configurables de los experimentos: los mapas, los parámetros *alpha* y *gamma* del algoritmo *Q-Learning*; y las distintas estrategias de recompensa.

5.1 Mapas

En este dominio de laberintos lógicos hemos diseñado 3 mapas de dificultad creciente, que explicaremos a continuación:

5.1.1 Mapa fácil

El mapa fácil se hizo de tal forma que fuese un espacio vacío, sin obstáculos. Sirvió para realizar una primera prueba con el algoritmo.

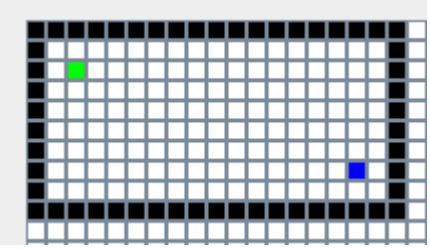


Figura 23. Mapa lógico fácil

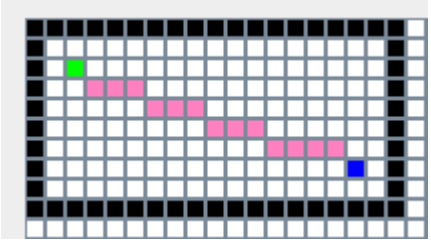


Figura 24. Mapa lógico fácil resuelto

En este mapa de dimensiones 17 x 8 casillas, tenemos que diferenciar cuatro tipos de casillas. Las casillas negras son infranqueables, sirven como muro. La casilla azul es la casilla inicial y verde es la final (la meta). Las casillas blancas son por las que se puede pasar.

Como se puede observar en la figura 23, este mapa no se puede llamar laberinto, ya que es un espacio abierto con forma rectangular y bastante pequeño. El camino más óptimo para realizar el recorrido entre la casilla inicial y la final es de 14 pasos (figura 24), aunque este camino se puede conseguir de varias maneras distintas. Dado que la dimensión de este mapa es tan pequeña (17x8), la partida se reinicia automáticamente si la unidad no ha alcanzado la meta tras 100 pasos.

5.1.2 Mapa medio

El mapa medio se hizo con la idea de que hubiese en un principio dos caminos similares hasta la meta: uno con trampas y el otro no. Así podíamos comprobar si el algoritmo aprendía la mejor opción.

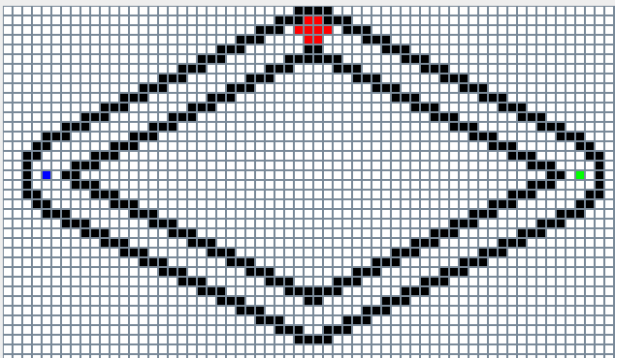


Figura 25. Mapa lógico medio

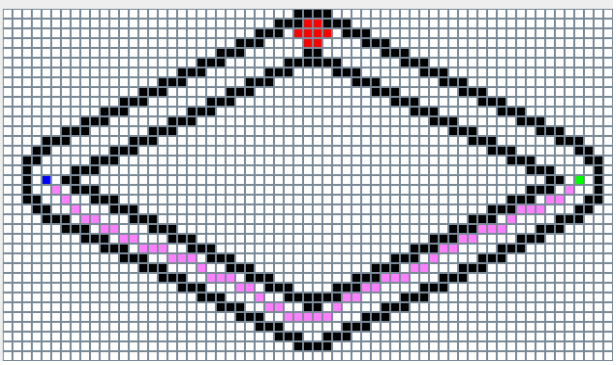


Figura 26. Mapa lógico medio resuelto

El mapa de la figura 25, de nivel medio, añade un nuevo elemento: las casillas rojas, que representan trampas que matan instantáneamente a la unidad. Este ofrece aparentemente dos caminos posibles para llegar a la meta, porque se debe mencionar que el algoritmo de aprendizaje no tiene forma de saber a priori qué camino es mejor. Como funciona a base de prueba-error, tendrá que explorar las dos posibilidades. Por lo tanto sólo existe una vía posible para llegar a la meta. El camino óptimo son 55 pasos (figura 26). Y sucede lo mismo que en el mapa fácil, puede haber muchos caminos óptimos de 55 pasos, y todos muy parecidos entre sí, con pequeñas variaciones del recorrido dependiendo de por donde se elija subir o bajar. En este mapa, como es más grande, y hay una bifurcación, el número máximo de pasos que permitimos en cada simulación es de 500.

5.1.3 Mapa difícil

En el mapa difícil generamos varios caminos posibles, con distintas trampas. Además en uno de los caminos hay que alejarse de la meta para poder llegar a ella. El objetivo era ofrecer un mapa más complejo de resolver.

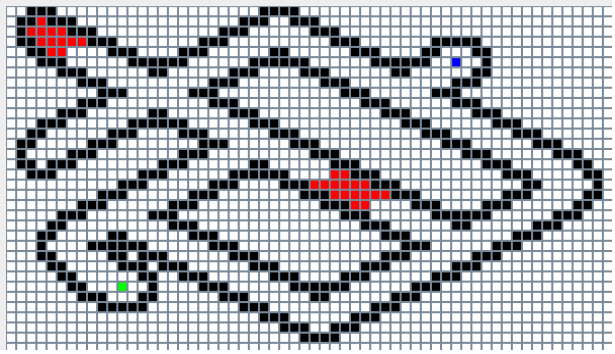


Figura 27. Mapa lógico difícil

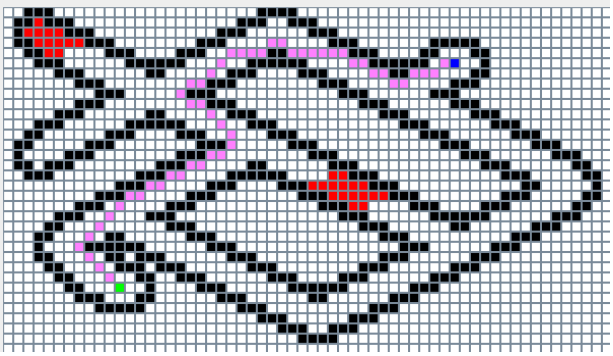


Figura 28. Mapa lógico difícil resuelto

El mapa difícil, como se ve en la figura 27, ya es un laberinto en sí. Esto va a dificultar la tarea de conseguir el camino óptimo (figura 28), ya que las posibilidades de llegar a la meta son más y el mapa es más grande. Tiene varios pasillos, algunos de ellos sin salida, y por primera vez dos caminos distintos para llegar a la meta. Uno es más largo que el otro.

Además, ocurre lo mismo que en los otros mapas, hay varias formas de conseguir el valor óptimo. Por la complejidad del mapa, el número máximo de pasos que permitimos en cada simulación es de 1500.

5.2 Experimentos con *Alpha* y *Gamma*

Como hemos mencionado, los parámetros configurables del aprendizaje en nuestro caso concreto eran tres: mapas, *alpha* y *gamma*, y estrategias de recompensa.

Por un lado decidimos hacer cada prueba con los 3 mapas. Dentro de cada mapa, había que probar los parámetros *alpha* y *gamma*, que al ser un conjunto de números infinitos comprendidos entre 0 y 1 decidimos reducirlo a 5 posibles valores de *alpha* y 5 de *gamma*. Por otro lado, por cada combinación de valores *alpha/gamma*, debíamos hacer pruebas con 4 posibles estrategias de recompensa. Todo esto hacía que el número de combinaciones fuera inabarcable (3 mapas * 5 *alpha* * 5 *gamma* * 4 estrategias = 300 posibles combinaciones), y por ello decidimos separar los experimentos, realizando por una parte las pruebas de *alpha/gamma* y por otro las de las posibles estrategias.

En este apartado vamos a mostrar los resultados de los experimentos con *alpha* y *gamma*. La estrategia de recompensa que se ha seguido en todos ellos es la de dar una recompensa positiva únicamente al llegar a la meta, y negativa en caso de muerte.

Para empezar, debemos recordar que la actualización de los valores de la *QTabla* sigue la siguiente fórmula (figura 29):

$$Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$$

Figura 29. Fórmula *Q-Learning*

Como hemos indicado en el apartado 3.3.2 “Parámetros *Alpha* y *Gamma*”, *alpha* representa el ratio de aprendizaje (o la importancia de la recompensa recién adquirida), mientras que *gamma* representa el factor de descuento (o importancia de las recompensas futuras). Partiendo del hecho de que ambos se deben encontrar entre 0 y 1, y que se suelen configurar de forma experimental, lo primero que hicimos fue probar cada uno de los tres mapas con distintas combinaciones de *alpha* y *gamma*. Decidimos probar únicamente con 5 valores para cada parámetro: {0,1; 0,3; 0,5; 0,7; 0,9}, porque la

ejecución de cada uno de estos experimentos requiere varias horas. A continuación vamos a mostrar y comentar los resultados para los tres mapas.

5.2.1 Mapa fácil

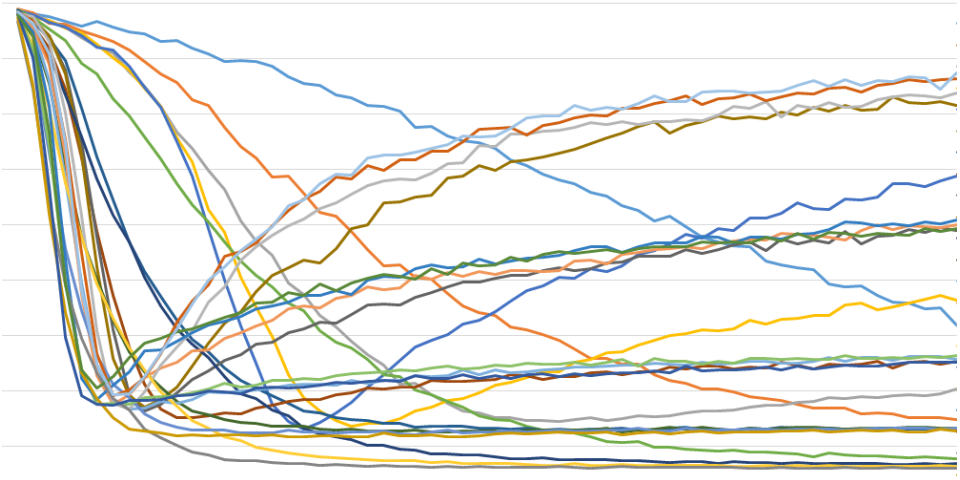


Figura 30. Cálculos de *alpha* y *gamma* en el mapa fácil

La gráfica de la figura 30 representa el número de pasos que se han dado hasta completar el laberinto en una serie de simulaciones. El eje X es el número de simulaciones completas que se han ejecutado con la misma *QTabla*, y el eje Y es el número de pasos que se han dado hasta completar el laberinto en cada simulación. Esta gráfica muestra como el valor de *alpha* y *gamma* afectan en gran medida al aprendizaje del algoritmo. A continuación analizaremos los resultados por separado.

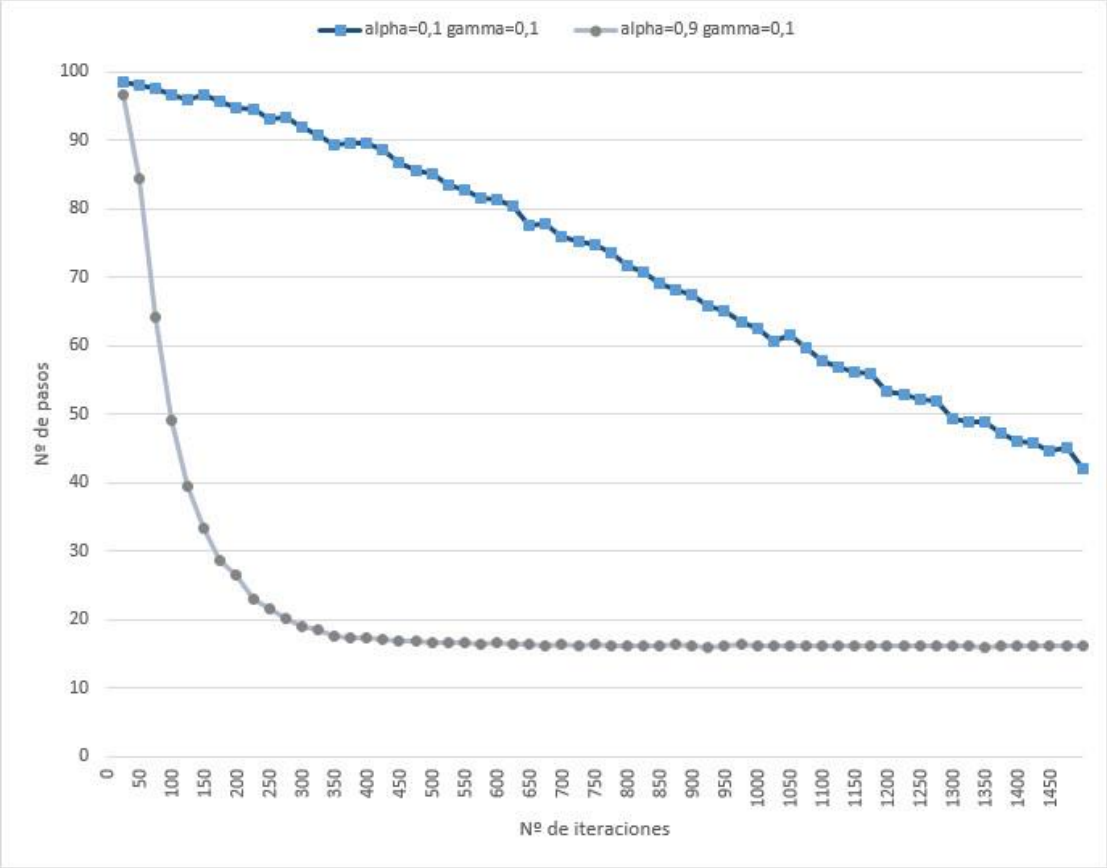


Figura 31. Comparativa del mapa fácil, *mejor y peor resultado*

En la figura 31, si comparamos los resultados de la ejecución con *alpha* = 0,1 y *gamma* = 0,1 (la de arriba), con los resultados de *alpha* = 0,9 y *gamma* 0,1 (la de abajo), podemos observar, que una aprende mucho más rápido que la otra. Se puede observar que con apenas 300 iteraciones la mejor ya ha aprendido la política óptima, y con 1500 iteraciones la ejecución de *alpha* = 0,1 y *gamma* = 0,1 todavía no ha aprendido un camino cercano al óptimo, quedándose por encima de los 40 pasos. Esto nos indica, como norma general, que los valores de *alpha* más pequeños hacen que el agente aprenda más despacio.

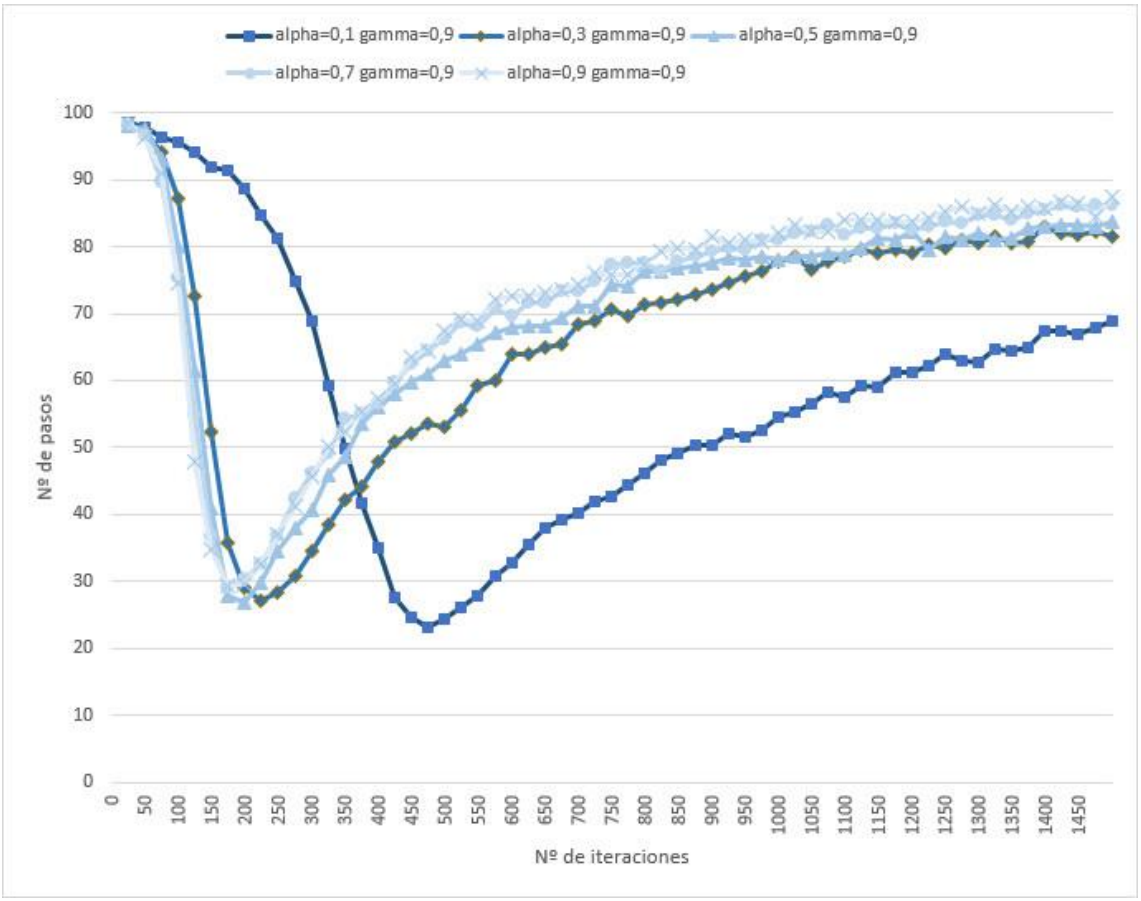


Figura 32. Comparativa del mapa fácil, *gamma* 0,9

En esta gráfica (figura 32) se puede observar que hay ciertos valores, que aprenden rápido, pero luego empiezan a recorrer caminos más largos.

Hemos seleccionado aquellas ejecuciones con *gamma* = 0,9, ya que son en las que este efecto es más marcado, pero se produce un efecto similar (en menor o mayor medida) en todas las que *gamma* >= 0,5; todas resultaban muy parecidas. Esto es porque se le está dando excesiva importancia al contenido de la *QTabla* del estado al que se accede, con independencia de cómo se haya llegado al mismo. Esto implica que al principio puede empezar a encontrar y reforzar buenos caminos, con lo que va acumulando un valor muy grande en dicho camino. Pero como nuestra política de selección de acciones tiene un factor de aleatoriedad, en las iteraciones que explore caminos equivocados (por ejemplo, moviéndose en dirección contraria), se estará reforzando fuertemente esta

acción al tener un *gamma* muy alto. Esto hace que se refuercen en gran medida las equivocaciones (en este caso, avanzar hacia atrás). Además, En este caso, y de acuerdo a lo que hemos dicho anteriormente, se ve que cuando *alpha* es cercana a 0, el aprendizaje (y posterior “des-aprendizaje") son más lentos que cuando *alpha* es cercana a 1.

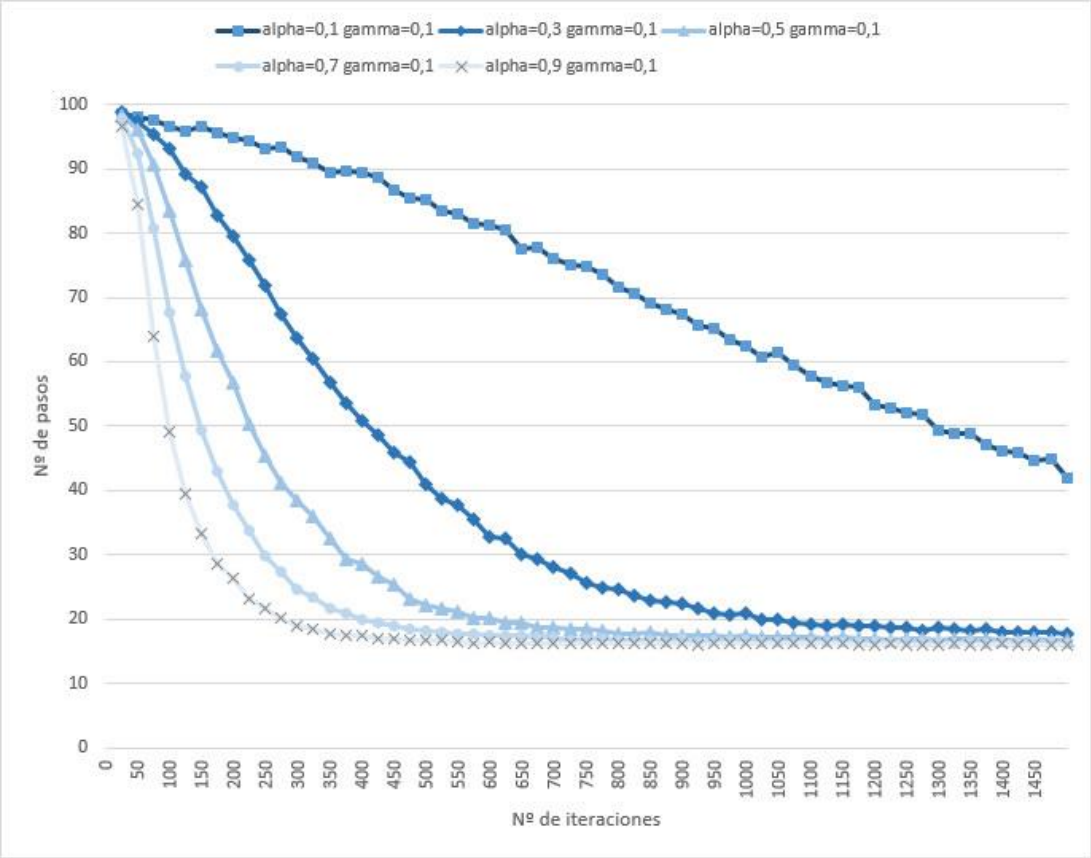


Figura 33. Comparativa del mapa fácil, *gamma* 0,1

Por último, en la gráfica de la figura 33 hemos aislado las ejecuciones en las que *gamma* es cercana a 0. Cabe destacar que todos los experimentos donde *gamma* = 0,1 son las mejores opciones, y en concreto, la que tiene el *alpha*=0,9 es la que más rápido aprende y consigue un mejor valor mínimo. La selección de *gamma* cercana a 0 es buena porque no prioriza las recompensas futuras. Esto puede hacer que el aprendizaje sea ligeramente más lento, pero a la larga será más fiable.

5.2.2 Mapa medio

Con el mapa de nivel medio, los resultados son similares, aunque con algunas excepciones. Aquí mostramos todos los resultados con todas las combinaciones de *alpha* y *gamma* posibles.

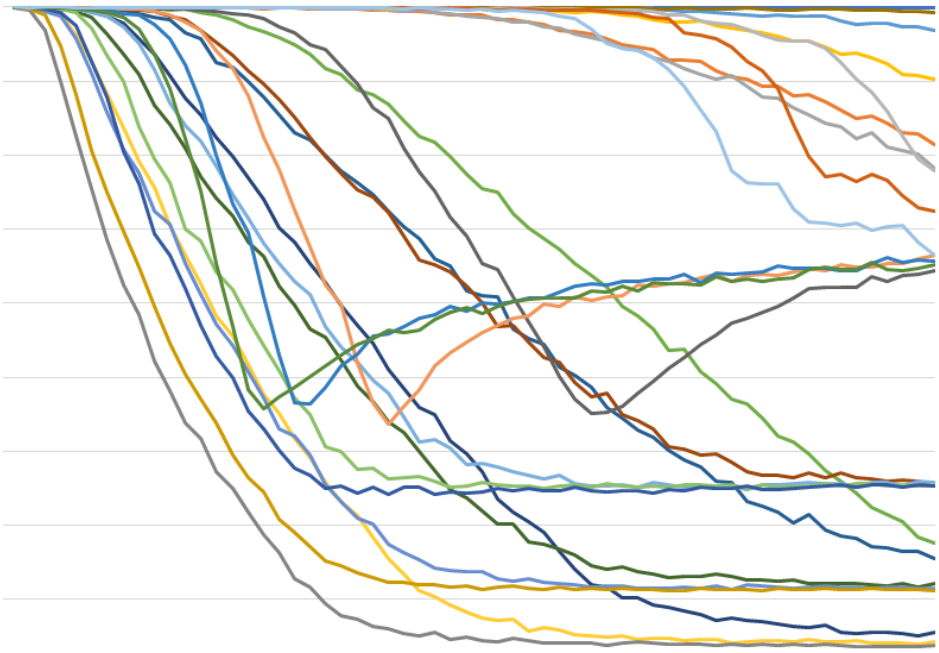


Figura 34. Cálculos de *alpha* y *gamma* en el mapa medio

Como se puede observar en este gráfico (figura 34), pasa lo mismo que en el mapa fácil: hay unos valores de *alpha* y *gamma* mucho mejores que otros. Asimismo, se percibe que hay gráficas que tienen valores con una tendencia positiva (correspondientes a las *alpha* cercanas a 1) y otras que empiezan descendiendo pero luego ascienden (correspondientes a un valor elevado de *gamma*). También se puede ver, en todas las funciones en general, que la velocidad de aprendizaje es menor que en el mapa fácil. Con todo esto, hay un conjunto de funciones más aisladas en la parte superior derecha de la gráfica, que al ser una excepción, procederemos a analizar.

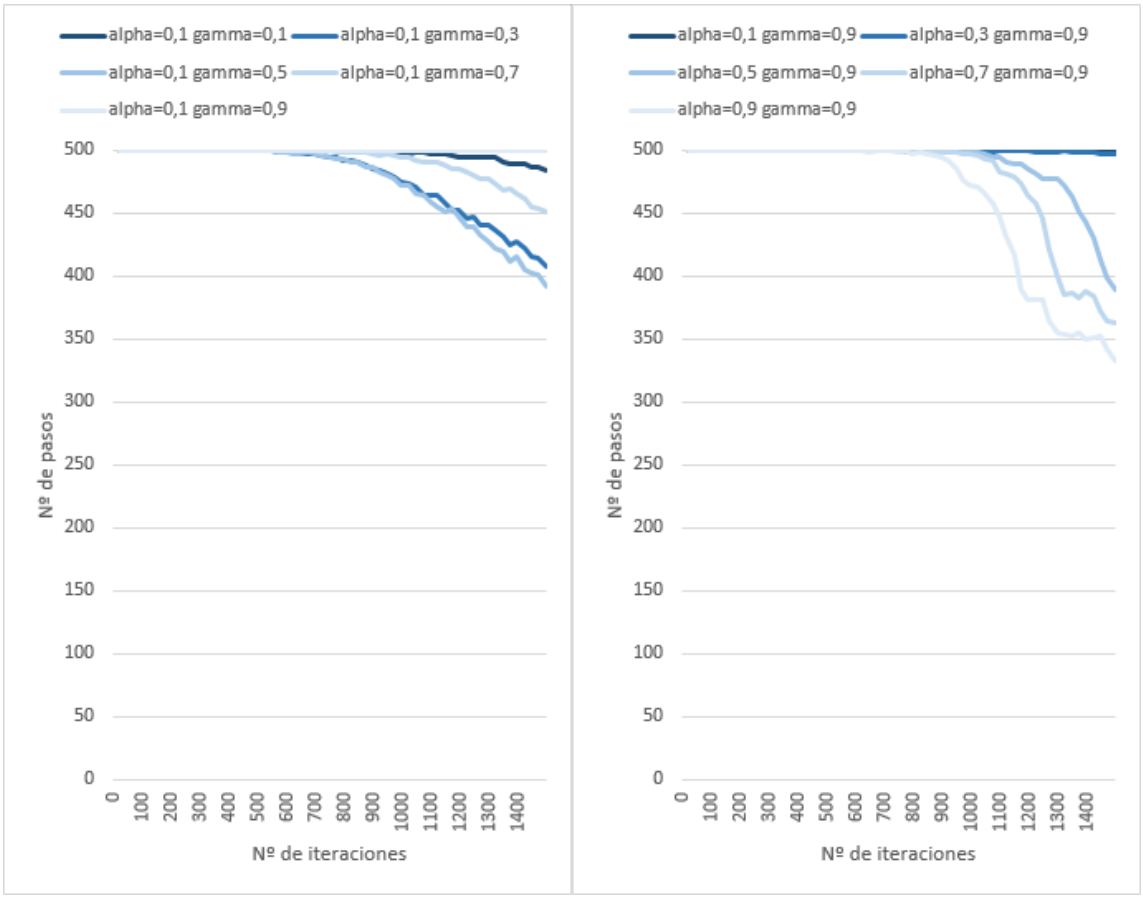


Figura 35. Comparativa del mapa medio, *alpha* 0,1 y g*amma* 0,9

Las dos gráficas de la figura 35 representan las funciones con valores de *alpha* cercanos a 0 (a la izquierda), y con *gamma* cercana a 1 (a la derecha). Estas funciones suponen una excepción ya que se encuentran algo más separadas del resto.

En la gráfica de la izquierda se encuentran las funciones con *alpha* cercana a 0. Esto significa que el aprendizaje será mucho más lento, dando poca importancia a la información recién adquirida, por lo que tiene sentido que todas ellas empiecen a aprender pasadas ya las 500 iteraciones.

En la gráfica de la derecha, el factor de descuento (*gamma*) cercano a uno hará que el algoritmo se esfuerce por una alta recompensa a largo plazo. Esto, en sí mismo, no tiene porqué ser negativo, pero en este caso hace que la velocidad de aprendizaje sea notablemente menor. Esto se debe a que en cada uno de los pasos en los que la recompensa es 0 (que en la estrategia básica son la mayoría) la segunda parte de la

fórmula de *Q-Learning* ($\alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$) se anula prácticamente, por lo que el aprendizaje en los pasos intermedios se vuelve muy lento. Hay que aclarar que el mecanismo de aprendizaje de *Q-Learning* no sólo tiene en cuenta las recompensas positivas, si no que si no recibe recompensa poco a poco se va disminuyendo el valor en la *QTabla* y esto hace que se vayan explorando más otros caminos.

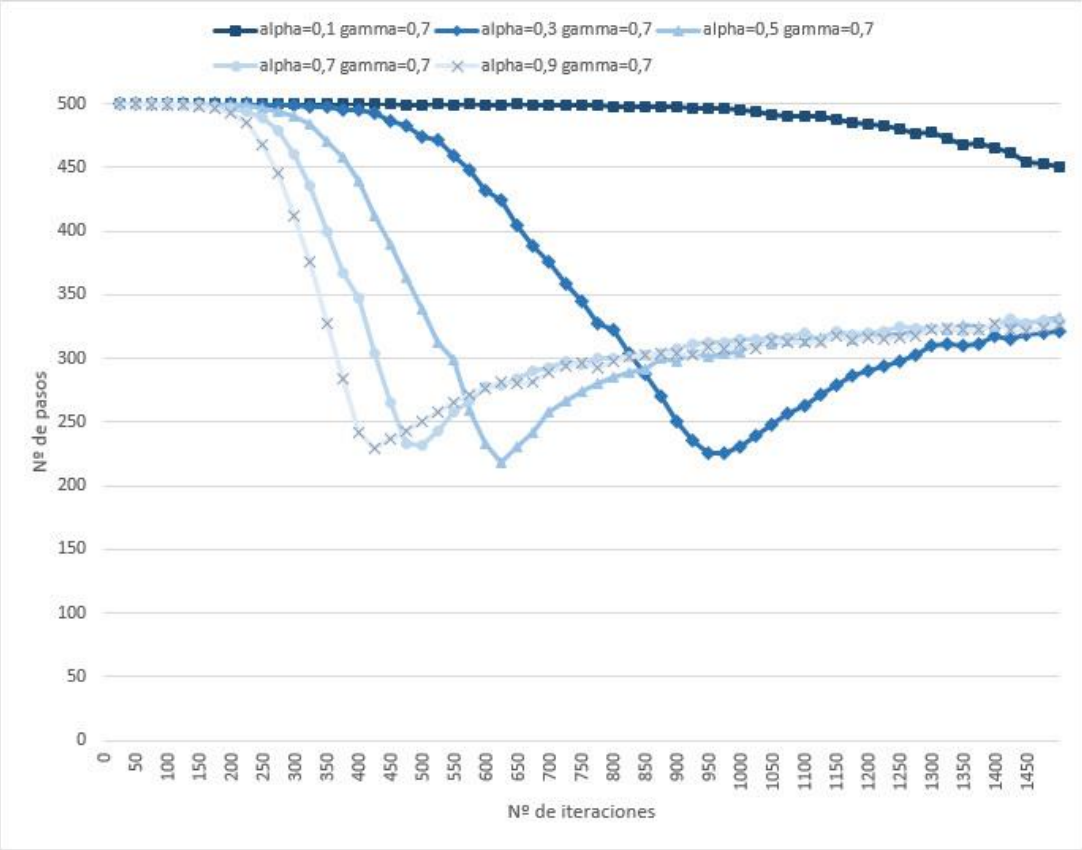


Figura 36. Comparativa del mapa medio, *gamma* 0,7

En la figura 36 se muestran aquellas ejecuciones que al principio aprenden, pero que después dejan de aprender. En este caso ocurre lo mismo que en el mapa fácil cuando *gamma* es cercana a 1, solo que en este caso esta tendencia se aprecia mejor para valores de *gamma* = 0,7. Como hemos explicado en el apartado anterior, un valor tan elevado de *gamma* puede recompensar equivocadamente la exploración de caminos

alternativos, cuando ya se ha encontrado y reforzado bastante un buen camino. Cabe destacar que para *alpha* = 0,1 la velocidad de aprendizaje es tan lenta que no se llega a ver el punto mínimo donde la función empieza a crecer. A pesar de ello podemos deducir que el comportamiento, antes o después, será el mismo que el del resto de ejemplos mostrados en esta gráfica.

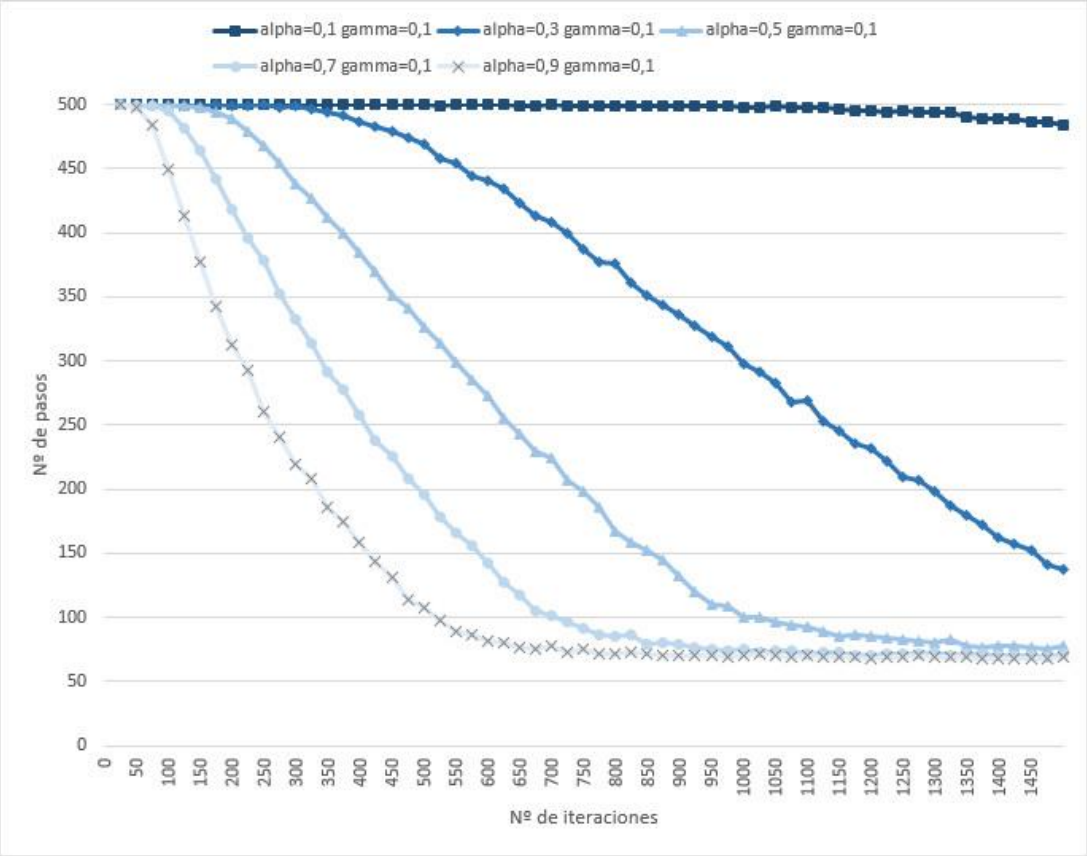


Figura 37. Comparativa del mapa medio, *gamma* 0,1

En la gráfica de la figura 37 se observan las ejecuciones donde *gamma* = 0,1. Entre ellas, obtienen un mejor resultado aquellas con mayor valor de *alpha*, en concreto la mejor es aquella en que *alpha* = 0,9 y *gamma* = 0,1. Esto se vuelve a deber a que un valor bajo de *gamma* hace que no se sobrevalore la recompensa futura, evitando el problema de la gráfica anterior de sobrevalorar los errores en la exploración, y un valor elevado de *alpha* hace que se tenga muy en cuenta la información recién adquirida, haciendo que la función aprenda rápidamente.

5.2.3 Mapa difícil

En la búsqueda de los valores de *alpha* y *gamma* adecuados para el mapa difícil, hemos tenido resultados similares al mapa fácil. Aquí mostramos las gráficas más relevantes.

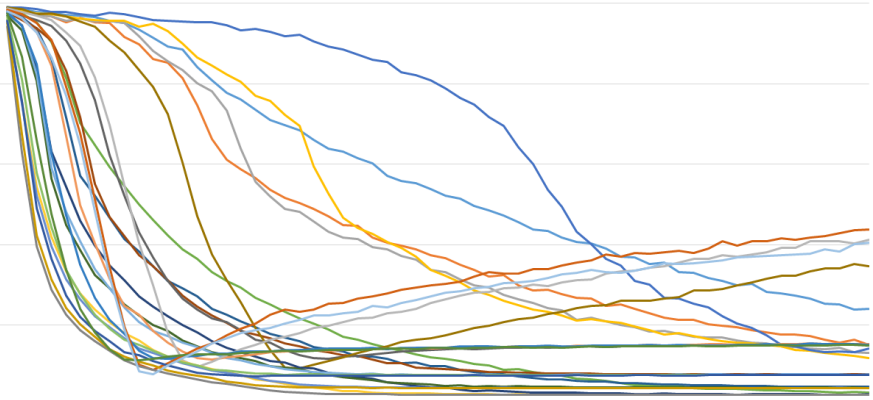


Figura 38. Cálculos de *alpha* y *gamma* en el mapa difícil

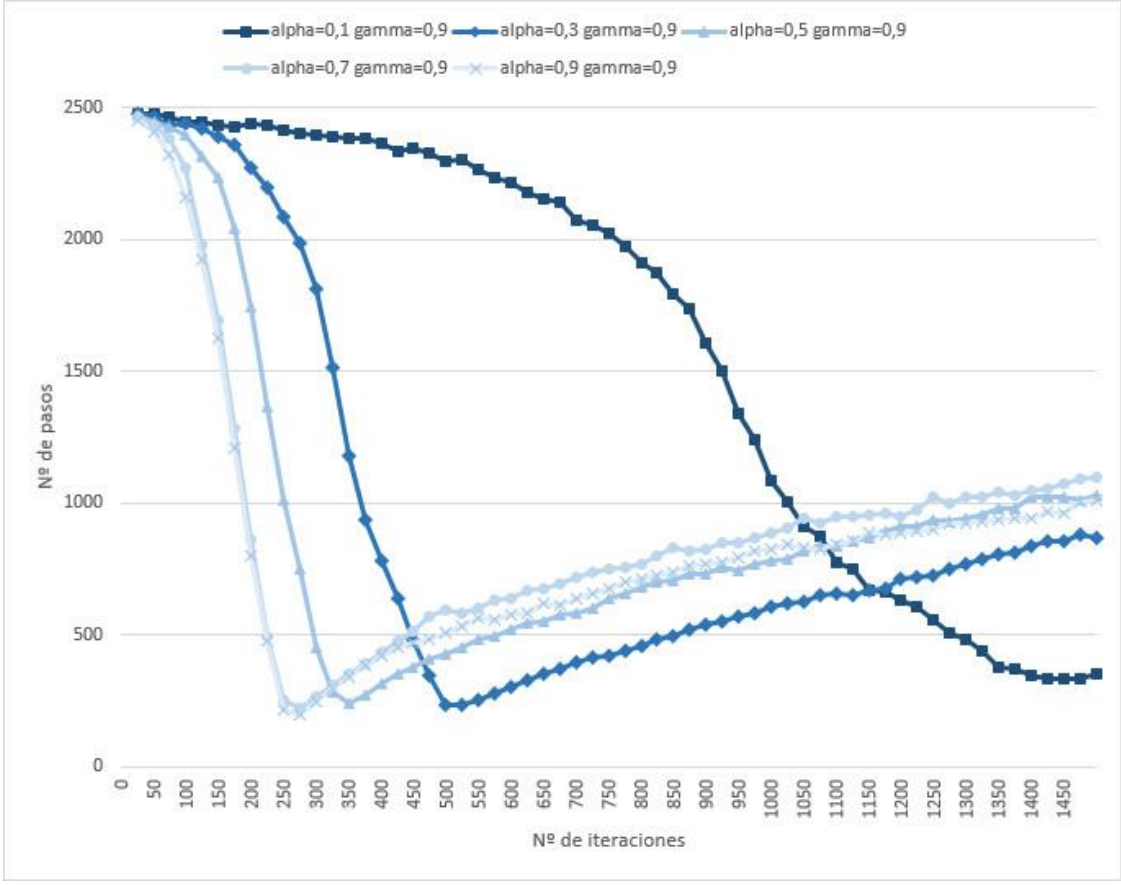


Figura 39. Comparativa del mapa difícil, *gamma* 0,9

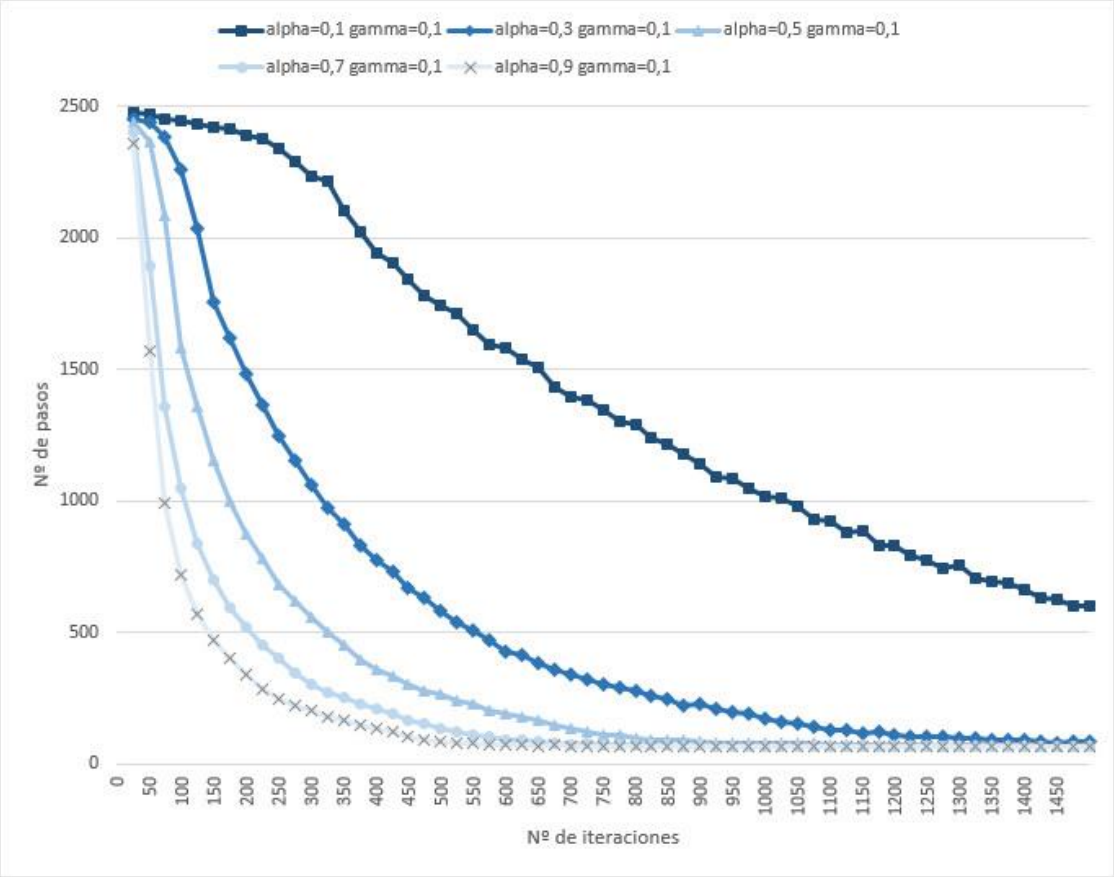


Figura 40. Comparativa del mapa difícil, *gamma* 0,1

En la primera gráfica (figura 38) volvemos a apreciar que existen ejecuciones notablemente superiores que otras, y esto depende de las combinaciones de *alpha* y *gamma*.

En la gráfica de la figura 39 (la segunda) volvemos a apreciar que las gráficas con valores de *gamma* muy elevados (0,9) siguen la tendencia de aprender y luego empezar a reforzar caminos más largos, muy posiblemente por la sobrevaloración de la información almacenada en la *QTabla*; la misma razón expuesta en los apartados anteriores.

Por último, en la figura 40 podemos apreciar los mejores valores para el mapa difícil, que otra vez vuelven a ser los valores con *gamma* cercana a 1, y tienen una curvatura de aprendizaje muy buena. Otra vez, el mejor valor de todos es el que tiene *alpha* = 0,9 y *gamma* = 0,1. A pesar de ello, para estos valores, se percibe que el número de iteraciones en alcanzar una política cercana a la óptima es algo superior en comparación

con el mapa fácil; en el fácil tardaba unas 400 iteraciones en encontrar una política cercana a la óptima, y en el difícil unas 500/550, debido a la mayor complejidad del mapa.

5.2.4 Conclusiones

Hemos comprobado que en nuestro caso los mejores valores son aquellos que tienen un valor de *alpha* muy alto y un valor de *gamma* muy bajo. Concretamente, hemos decidido escoger los valores de *alpha* = 0,9 y *gamma* = 0,1; que utilizaremos en el resto de nuestros experimentos. Esto se debe a que, como puede observarse en las distintas gráficas, son los valores que ofrecen el aprendizaje más rápido, pero a la vez más fiable y constante a largo plazo.

Por otro lado, resulta que los experimentos en los que *gamma* era bastante elevada (*gamma* >= 0,5 en el mapa fácil y *gamma* >= 0,7 en los mapas medio y difícil) eran justo los resultados en los que la función tenía un comportamiento anómalo: empezaba aprendiendo pero al final retrocedía y empeoraba su resultado. Esto viene provocado porque si se le da un valor excesivamente elevado a *gamma*, una vez se ha encontrado un camino bastante positivo se pueden estar reforzando los errores producidos por la exploración aleatoria, haciendo que estos pasos equivocados adquieran una gran valoración instantáneamente. Es bastante probable que con otra política de selección de acciones (de máximos) no se obtuviera este comportamiento, ya que no se explorarían caminos de manera aleatoria.

También podemos afirmar que, como norma general, los valores de *alpha* más altos hacen que el agente aprenda más rápido, y viceversa.

Por último, cabe aclarar que estos valores sirven únicamente para estos experimentos. Esto se debe a que estos resultados han sido obtenidos siguiendo una estrategia de recompensa básica (que sólo asigna recompensa en la meta), y una política de selección de acciones probabilística. Si modificáramos tanto una como la otra, los resultados obtenidos serían distintos.

5.3 Experimentos con estrategias de recompensa

Una vez hallados los mejores valores de *alpha* y *gamma* nos queda hacer experimentos con cuatro distintas estrategias de recompensa.

La primera es la **estrategia básica**, que es la que hemos utilizado en los experimentos de *alpha* y *gamma*. Está sólo da recompensa en dos casos: cuando llega a la meta (recompensa positiva) y cuando muere (recompensa negativa). En el resto de los casos la recompensa es 0.

La **estrategia euclídea** es similar a la básica, pero se diferencia en que a cada paso (que no gana ni pierde) también da una recompensa. Esta recompensa se asigna si se está acercando a la meta y es proporcional a la distancia a la que se encuentra de la misma.

La tercera estrategia, ***"less steps"***, se diferencia de la básica en que en vez de dar una recompensa constante cuando llega a la meta, da una recompensa inversamente proporcional al número de pasos que haya tardado en llegar (más recompensa cuanto menos pasos emplee).

A la última estrategia la hemos llamado ***"euclidean distance and less steps"***, y esto se debe a que intenta integrar las dos anteriores. Por un lado recompensa cada paso siguiendo la estrategia euclídea, y por otro da una recompensa dependiente del número de pasos empleados al llegar a la meta.

Además de estas cuatro, probamos una última estrategia a la que llamamos ***"repeated states"***, que intentaba penalizar al agente cuando pasaba dos veces por un mismo estado, pero en los primeros experimentos que realizamos los resultados no fueron muy positivos, por lo que decidimos abandonarla.

Una vez descritas las estrategias hay que decidir qué valores emplear en las recompensas. Para las **recompensa al llegar a meta**, y mediante algunas pruebas experimentales, comprobamos que con valores muy bajos (10, 50, 100…) no aprendía muy rápido; en cambio, con valores muy altos (10^8, 10^9, 10^{10}...) no exploraba suficientes caminos. Por ello elegimos un valor (1.000) que experimentalmente funcionaba bien.

Para la **recompensa al morir** sucedía algo similar: una recompensa muy negativa impediría volver a elegir esa acción con sólo fallar una vez, pero si no era lo suficientemente negativa no tendría ningún efecto en la política. Al final escogimos el valor de -1.

Para la **recompensa de la estrategia euclídea** decidimos que cada paso que se acercara a la meta daríamos un refuerzo entre 0 y 1 proporcional a la distancia a la meta. Esta recompensa es tan pequeña porque se recibe a cada paso, y no tenía que anular el efecto de la "gran" recompensa final.

La **recompensa de la estrategia *less steps*** debía ser un valor que dependiera del número de pasos empleados en llegar a la meta. Por eso buscamos una función que relacionase los pasos dados con la recompensa recibida. Por ejemplo, podría representarse mediante una gráfica como la de la figura 41, en la que al principio la recompensa es máxima, y va decreciendo rápidamente.

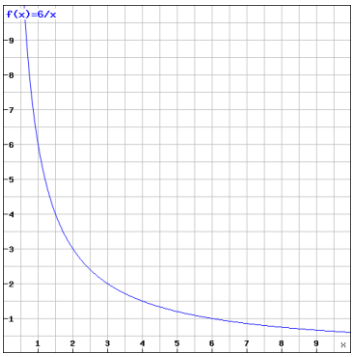


Figura 41. Función de recompensa en la estrategia *less steps*

Por último, las recompensas de ***euclidean distance and less steps*** son las correspondientes de las dos estrategias anteriores.

En el diagrama de la Figura 42 podemos ver el funcionamiento de esta última estrategia, que combina varias:

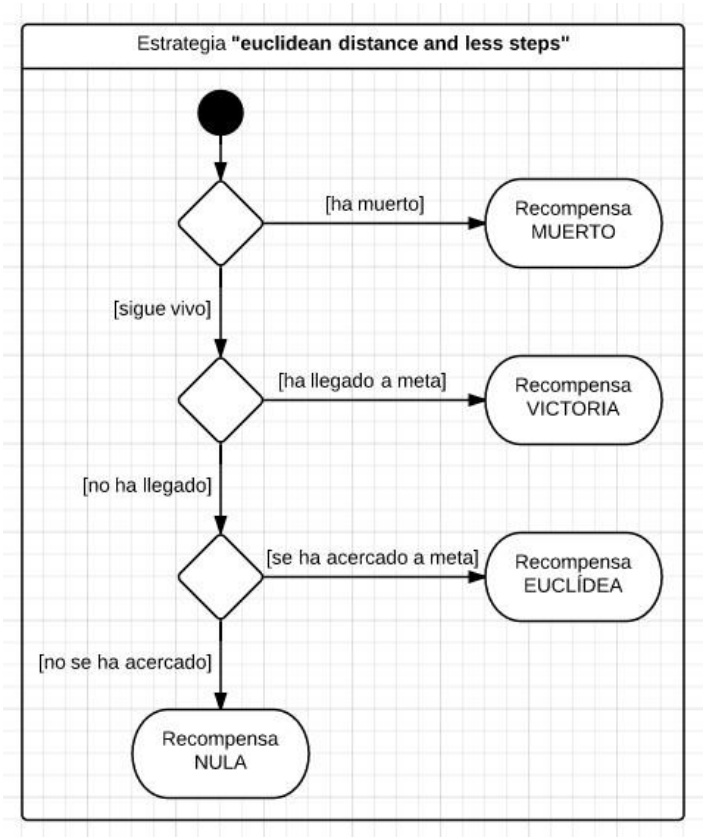


Figura 42. Diagrama de estados para la estrategia *euclidean distance y less steps*

Esta estrategia no tuvo mucho éxito. Pudiera parecer que el concepto de juntar varias estrategias en una sola, ajustando el peso de cada una, debería proporcionar un aprendizaje más rápido y por tanto más potente. Pero las interacciones entre las distintas recompensas parece que confundían más que ayudaban al algoritmo de aprendizaje.

En las siguientes páginas vamos a mostrar los resultados de ejecutar las distintas estrategias en los tres mapas.

5.3.1 Mapa fácil

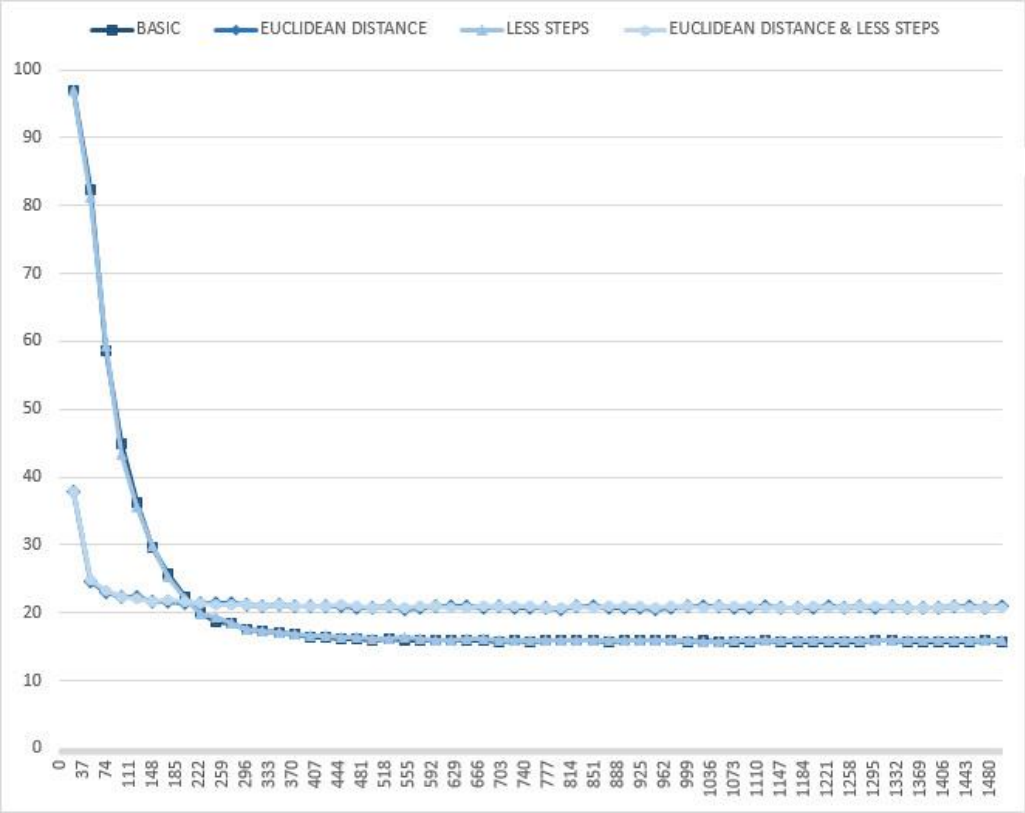


Figura 43. Comparativa del mapa fácil, estrategias

En la gráfica de la figura 43 se puede observar que la estrategia básica y la estrategia de menos pasos son muy parecidas, y que la estrategia euclídea y la de euclídea con menos pasos también. Esto se debe a que en el fondo son la misma estrategia, solo que al llegar a la meta obtienen una recompensa mayor o menor. Por ello vamos a hablar directamente de estrategia euclídea y estrategia básica.

Se puede ver que la estrategia euclídea aprende mucho más rápido que la básica. Esto se debe a que el mapa fácil es totalmente abierto y no tiene ningún muro entre las posiciones inicial y final. Por ello al principio recompensa rápidamente el hecho de acercarse a la meta y por ende llega antes. Viendo la estrategia básica, observamos que al final es la que obtiene un mejor valor. Esto se puede deber a que sólo recompensa al llegar a la meta, por lo que no se establece un sesgo en la forma de alcanzarla. En este

caso concreto se puede ver que para un número de iteraciones menor a 200 (punto de corte entre ambas) la estrategia euclídea obtiene mejores resultados, y para más iteraciones es la estrategia básica la que mejor funciona. Esto puede ser interesante, de tal forma que si vamos a realizar menos de 200 iteraciones en el proceso de aprendizaje, nos puede interesar coger la euclídea.

Por último, queda añadir que con la estrategia básica se consigue alcanzar un valor cercano al óptimo, el cual es de 14 pasos.

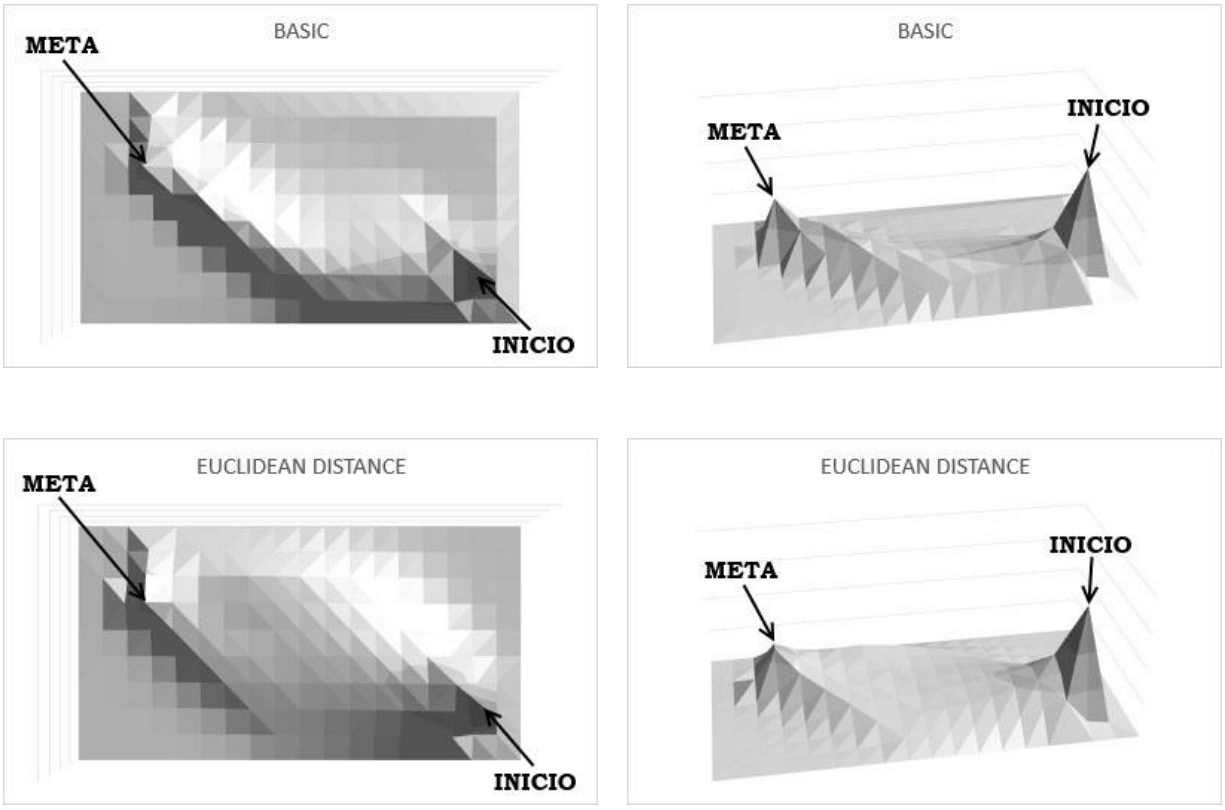


Figura 44. Caminos más visitados del mapa fácil

Las gráficas de la figura 44 representan los caminos más visitados. En estos gráficos de superficie, la altura de cada casilla representa el número de veces que la unidad ha pasado por esa casilla para alcanzar la meta. Se puede observar que, efectivamente, el camino que suele hacer es similar a uno de los posibles caminos óptimos.

En las gráficas de la estrategia euclídea se puede percibir que los caminos visitados son más dispersos que en la estrategia básica, que tiene un camino mucho más marcado. Esto se puede deber a que la estrategia euclídea prioriza “avanzar hacia la meta”, por lo que no es muy relevante qué camino tome (todos son buenos), mientras que avance hacia la meta. En cambio la estrategia básica carece de toda información del entorno, ya que sólo recibe recompensa al final, por lo que una vez que encuentra un camino que llega a la meta lo va reforzando.

5.3.2 Mapa medio

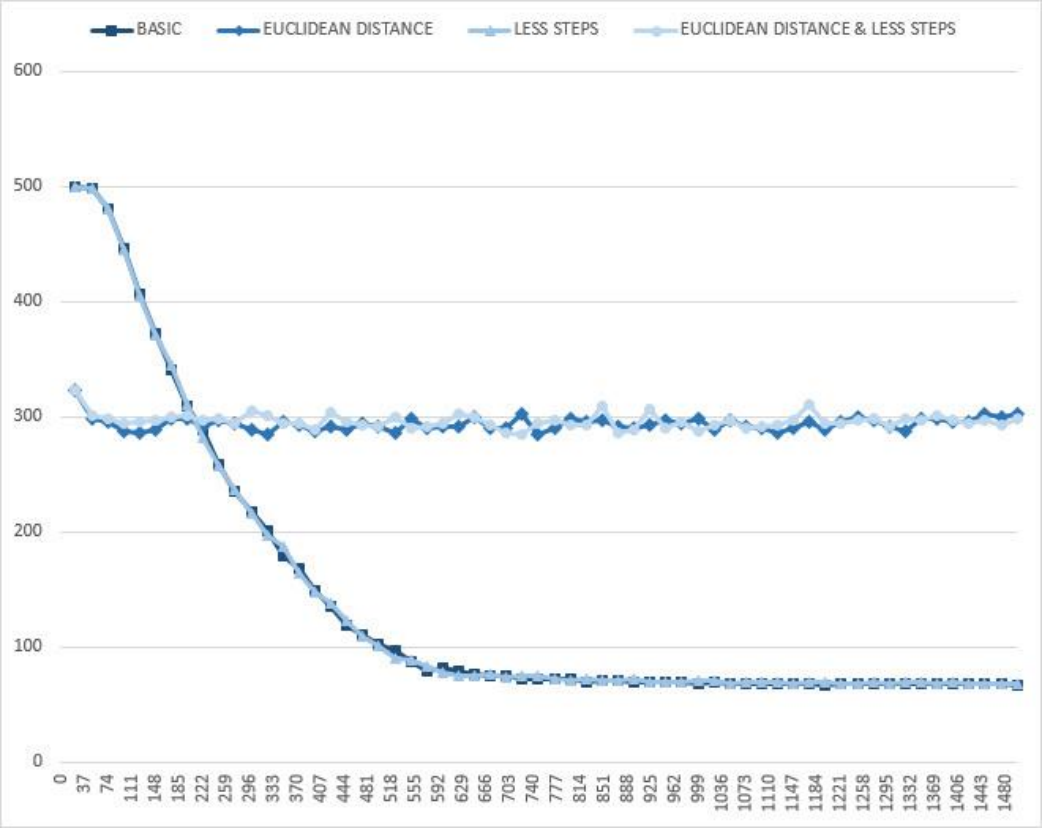


Figura 45. Comparativa del mapa medio, estrategias

En esta gráfica (figura 45) se puede apreciar lo mismo que en la anterior: las estrategias básica y *“less steps”* son bastante parecidas, al igual que las otras dos. Cabe destacar que esta vez en el mapa medio (recordando que era un pasillo con forma de rombo) la estrategia euclídea es bastante mala, esto se debe a que esta estrategia refuerza en

cada paso por el hecho de acercarse a la meta, y puede darse el caso de que se esté acercando por el camino de las trampas (y recompense este camino). Al final estará avanzando por uno u otro camino indistintamente, y morirá un 50% de las veces, por lo que al final no aprende cuál es el camino bueno. Sin embargo, la estrategia básica es buena pese a que tiene una pendiente menor que en el mapa fácil, debido probablemente a las dimensiones del mapa. También es interesante remarcar el punto de corte entre ambas (que indica a partir de qué punto nos interesa elegir una u otra), que se da a las 225 iteraciones. Por último, queremos añadir que se consigue un valor cercano al óptimo, que son 55 pasos.

A continuación, en la figura 46 mostramos las gráficas que muestran los caminos más usados por el agente.

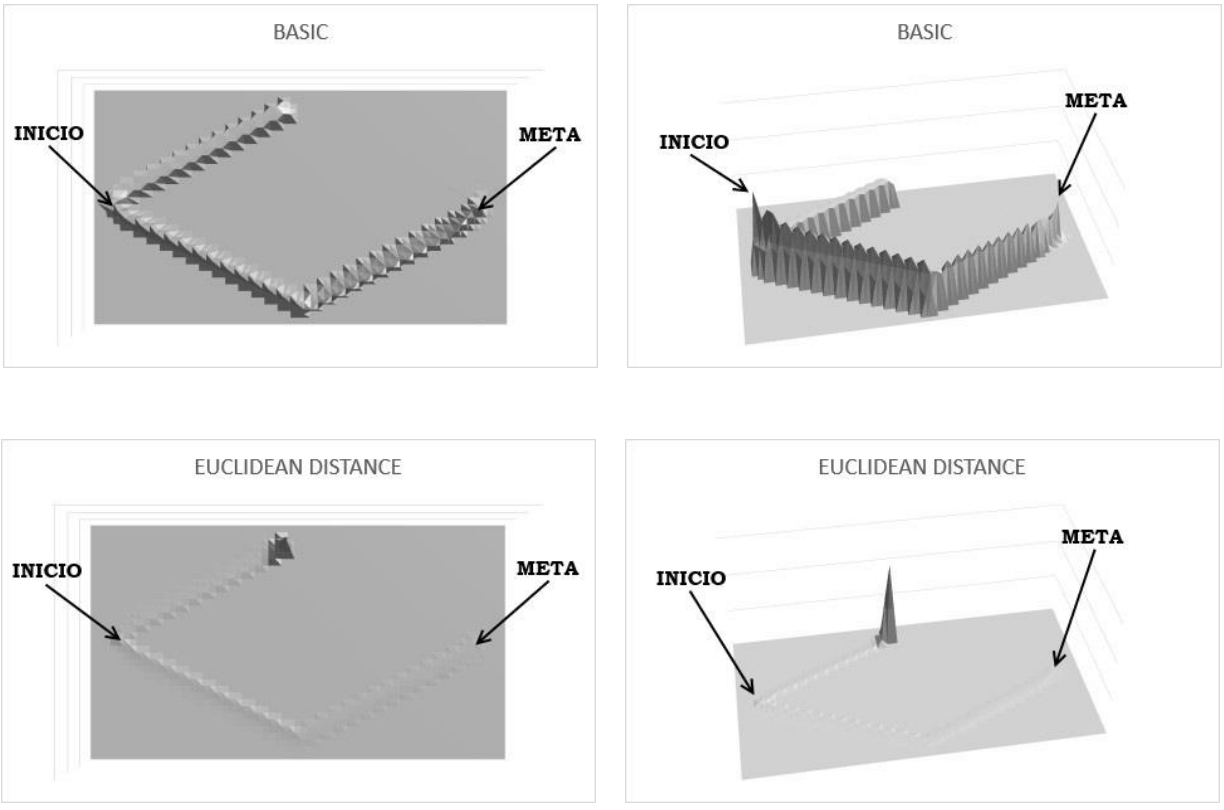


Figura 46. Caminos más visitados del mapa medio

En la estrategia básica se puede apreciar que visita ambos caminos, pero a medida que pasa el tiempo aprende que por abajo obtiene una recompensa, y por eso termina

obteniendo valores mayores en el camino de abajo. A pesar de ello, ha tenido que probar muchas veces ambos caminos para aprender cuál es el mejor.

La estrategia euclídea destaca que los valores de ambos caminos son bastante bajos y constantes (excepto unos pasos antes de donde se encuentran los enemigos). Esto se puede deber a que en un primer momento la estrategia euclídea "fuerza" a avanzar al agente hacia adelante (indistintamente del camino), por lo que le confunde y acaba muriendo la mitad de las veces. En las implementaciones habituales de *Q-Learning*, como es la nuestra, las recompensas negativas no se pueden propagar más de un estado hacia atrás, por lo que el agente no advierte que hay un enemigo hasta que es demasiado tarde [29] . Esto hace que se "bloquee", y no avance justo antes de llegar a la casilla del enemigo, por lo que pasa la mayor parte del tiempo en ese estado.

5.3.3 Mapa difícil

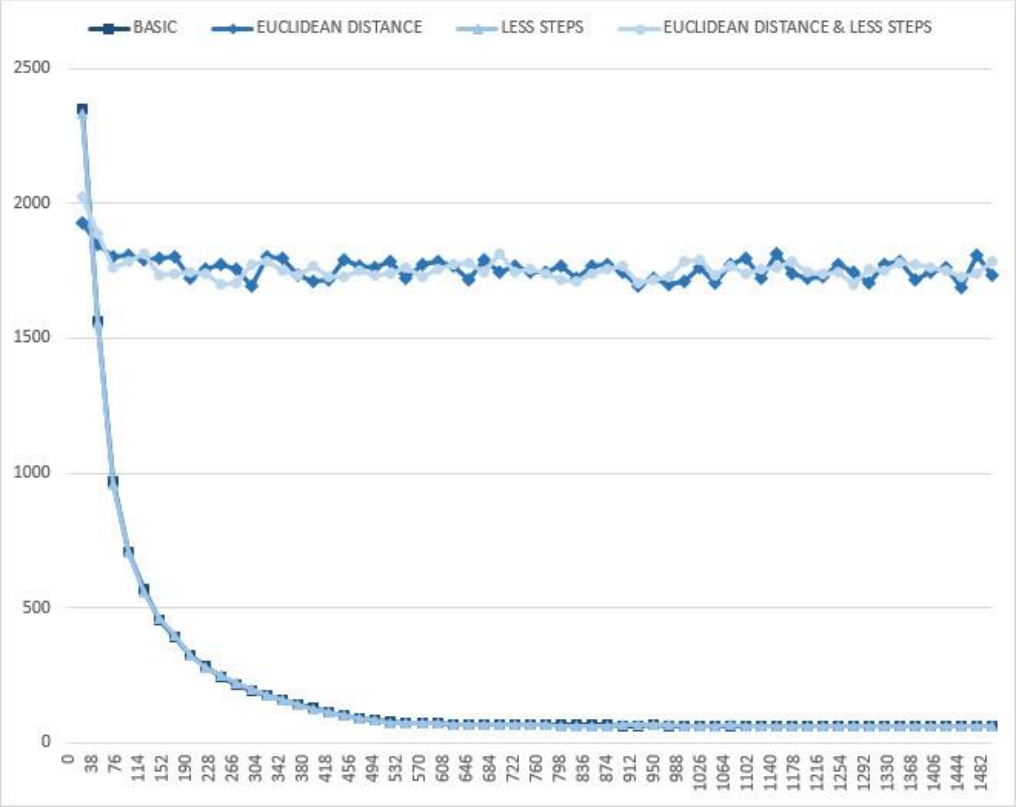


Figura 47. Comparativa del mapa difícil, estrategias

En esta gráfica, de la figura 47, se puede observar lo mismo que en la gráfica del mapa de nivel medio. La estrategia euclídea no es óptima porque llegado a un punto, se mete en un pasillo sin salida y se queda atascado, porque es la distancia más corta a la meta pero no hay camino, o muere en el intento. En cambio, la estrategia básica únicamente refuerza cuando llega al final, por lo que al final aprende el buen camino, aunque tarde bastante. Consigue un valor cercano al óptimo, que son 52 pasos. El comportamiento del agente se interpreta más fácilmente en las siguientes gráficas mostradas en la figura 48.

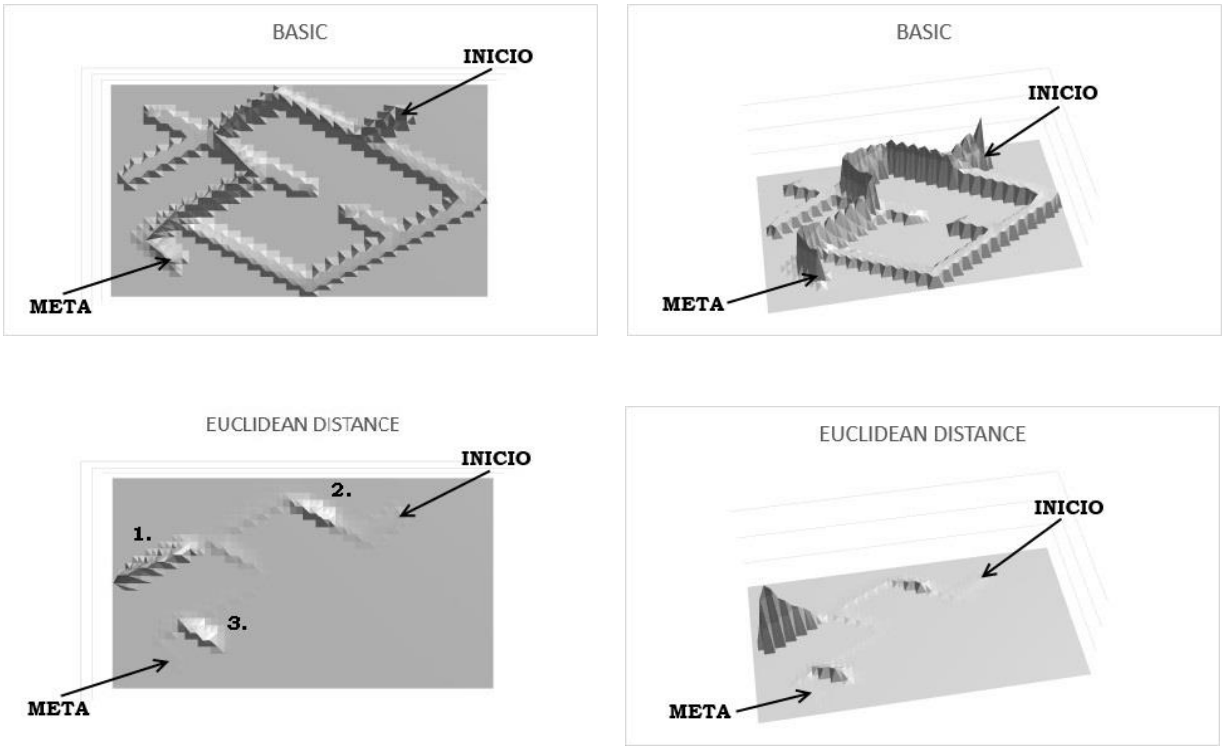


Figura 48. Caminos más visitados del mapa difícil

En la gráfica de la estrategia básica se puede apreciar, otra vez, como el reparto de estados visitados es mucho más equilibrado que con la estrategia euclídea. Esto se debe a que para encontrar la meta ha tenido que explorar todos los posibles caminos, sin ningún tipo de sesgo inicial. A pesar de ello, se puede ver que finalmente ha encontrado un camino mínimo, que queda más remarcado que el resto.

Por otro lado, podemos volver a comprobar que la estrategia euclídea "despista" al agente recompensando movimientos que no le llevan a ningún lado. En este caso podemos ver que permanece gran parte del tiempo bloqueado en un pasillo sin salida (número 1), porque siguiendo ese camino minimiza la distancia a la meta. Además hay otros 2 puntos conflictivos (números 2 y 3), en los que tiene que alejarse para continuar por el buen camino. Como esta estrategia recompensa acercarse a la meta, el agente prefiere no alejarse. A pesar de todo ello, podemos ver de forma muy difusa el recorrido del camino óptimo; esto significa que en gran número de ocasiones consigue llegar a la meta.

5.3.4 Conclusiones

Una vez realizados todos los experimentos sobre el mapa lógico podemos concluir lo siguiente:

Las estrategias básica y *"less steps"* resultaron ser superiores. Fueron muy positivas en los mapas fácil y medio, en un plazo de tiempo medio-alto (a partir de las 200 iteraciones). También sobre el mapa difícil en un plazo de tiempo muy corto (a partir de unas 25 iteraciones). Dichas estrategias son las mejores gracias a su simplicidad ya que solo recompensa positivamente cuando gana y negativamente cuando pierde. Además no emplean ningún tipo de conocimiento del dominio, y si se le permite realizar muchas iteraciones, encuentra el camino más ajustado a la solución óptima.

Por otro lado, la estrategia euclídea demostró no ser eficaz a la hora de alcanzar una mejor solución. Sin embargo, en el mapa fácil demostró ser capaz de aprender mucho más rápido que las anteriores en un corto periodo de tiempo, ya que eran caminos en línea recta hacia la meta. En cambio, en el medio aprendió que era bueno avanzar hacia la derecha (pero moría el 50% de las veces por una cuestión de probabilidad), y en el difícil intentaba acercarse a la meta a través de caminos sin salida, lo que le llevaba a bloqueos y caminos sin salida en multitud de ocasiones.

En vista de los malos resultados de la estrategia euclídea, la unión de esta con *"less steps"* no suponía en ningún caso una mejoría, ya que tan sólo reducía la recompensa al llegar a la meta.

Llegado a este punto trataremos de analizar cada mapa en más detalle, respecto a las dos estrategias más diferenciadas: la básica y la euclídea. Ya que las otras son prácticamente idénticas en resultados.

El mapa fácil, como su propio nombre indica, supone un aprendizaje rápido para la estrategia euclídea ya que no tiene ningún obstáculo para llegar a la meta y cualquier movimiento que le acerque a la misma supondrá en cualquier caso un avance, que se recompensará. Sin embargo, la estrategia básica tiende a coger las diagonales para "llegar antes" (por la importancia de la recompensa final). Pero en la euclídea, como recompensa prácticamente igual una diagonal que el avanzar en vertical o hacia la izquierda (y esta recompensa es mucho más importante que la recompensa final), va a tomar estos 2 últimos indistintamente (favoreciendo caminos más largos). O sea, que en vez de tomar la diagonal superior izquierda, toma primero arriba y luego izda. (o primero izqda. y luego arriba). Al hacerlo de este modo, en total da 5 pasos más (que coincide con la gráfica). Tomando la imagen del mejor camino en el mapa fácil:

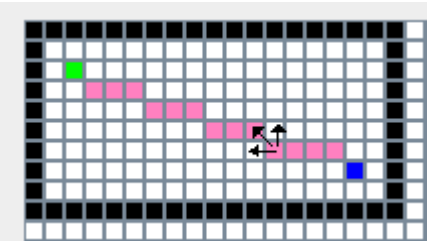


Figura 49. Posibilidades en la elección del camino óptimo

En la figura 49 se puede observar como existen 3 movimientos que serían recompensados positivamente con la estrategia euclídea, ya que se acercan a la meta.

La estrategia básica, al contrario, sólo explorará y recompensará caminos por los que haya llegado a la meta anteriormente, así que con las suficientes iteraciones conseguirá un camino mejor.

El mapa medio supone una toma de decisión para el algoritmo ya que las dos ramas son equidistantes de la meta. En él, la estrategia euclídea no supone ningún tipo de aprendizaje, pues acaba muriendo un gran número de veces.

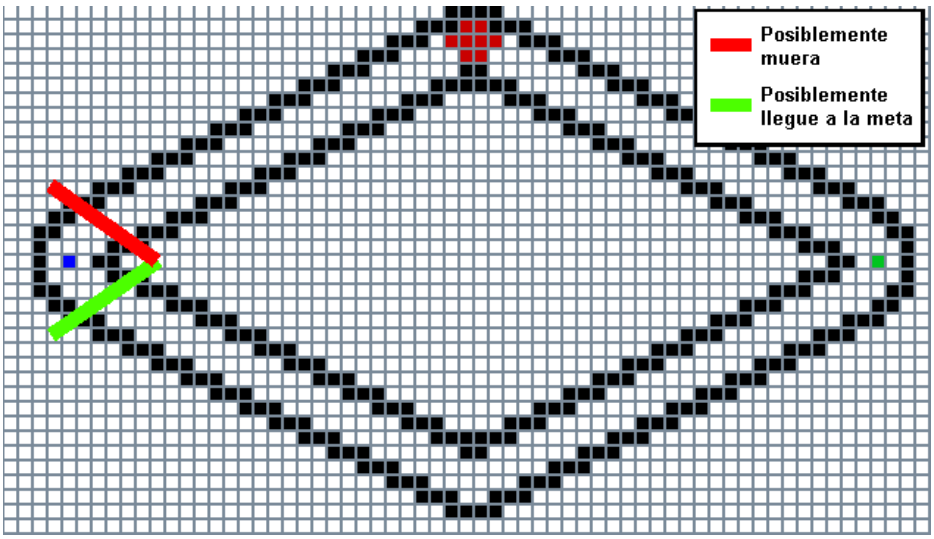


Figura 50. División de caminos en el mapa medio

En la imagen superior (figura 50) podemos observar cómo dependiendo de los primeros pasos, el resultado será llegar a la meta o morir, ya que siempre intentará caminar acercándose directamente a la meta. En este caso, el resultado es bien parecido al mapa fácil ya que en la estrategia euclídea, aunque elija inicialmente la rama superior, será recompensado positivamente aunque este camino le conduzca a una muerte segura. Mientras que la estrategia básica iría descartando la rama superior, en cuanto vaya reforzando el camino inferior, que le lleva a la meta y le da una recompensa positiva.

El mapa difícil es muy diferente en cuanto a sus resultados. La estrategia euclídea no supone ningún avance en un mapa donde el acercarse directamente a la meta supone meterse en un camino sin salida o quedarse bloqueado. Como bien muestra la imagen de la densidad de pasos en la figura 51.

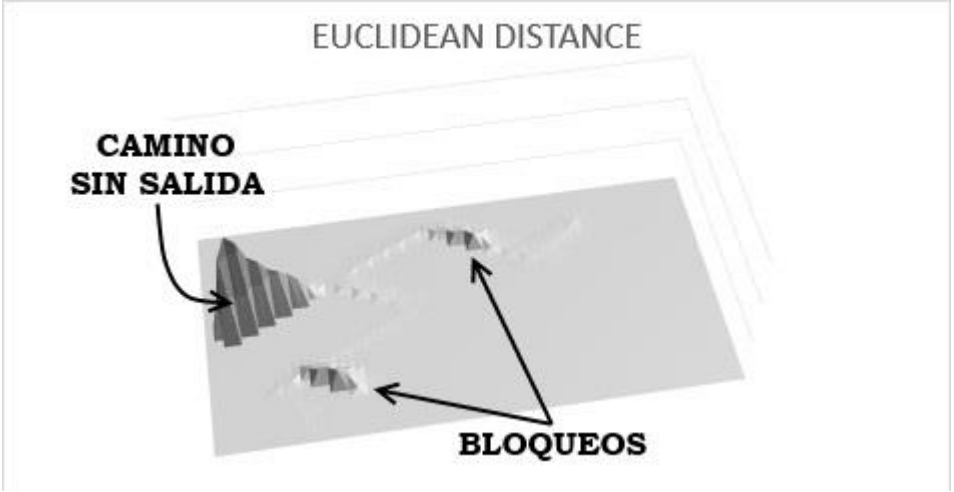


Figura 51. Bloqueos en mapa difícil, estrategia euclídea

De este modo buscar la solución que más nos acerca a la meta en cada momento no es para nada recomendable en un laberinto más complejo. Una vez más, la estrategia básica, con un número suficiente de iteraciones, demuestra que alcanza una solución más ajustada.

Capítulo 6. Resolución de laberintos en *StarCraft*

Una vez realizados todos los experimentos necesarios en el laberinto lógico es el momento de aplicar lo aprendido con otro dominio, el videojuego *StarCraft*.

Lo que hemos hecho en este capítulo, es repetir ciertos experimentos en tres mapas de *StarCraft* similares a los del laberinto lógico. Además, como partíamos de los resultados en el laberinto lógico, hemos utilizado los valores de *alpha*=0,9 y *gamma*=0,1 (que eran óptimos), y hemos realizado las pruebas únicamente con las estrategias básica y euclídea, que son las que se han demostrado más determinantes.

6.1 Cambiando el dominio

Gracias al *framework* de *Q-Learning*, hemos sido capaces de añadir un nuevo dominio para probar nuestro módulo de aprendizaje. Para configurar este nuevo dominio era necesario desarrollar un entorno propio de *StarCraft*.

StarCraft ofrece un dominio muy complejo y con mucha información que analizar en cada instante del juego. Es por ello que, no podemos estar almacenando toda la información en el entorno, es necesario modelar en el dominio almacenando, sólo aquella información que utilicemos. Así pues, en el *environment* de *StarCraft* había que decidir qué sería para nosotros un estado y cómo diferenciaríamos unos de otros. Nos decidimos por almacenar una tabla de estados basados en la posición lógica de la unidad terrestre (*marine*) dentro del laberinto.

A partir de ahora intentaremos explicar cuál ha sido el proceso de adaptación de nuestros mapas del laberinto lógico a mapas en *StarCraft*, y cuáles son los problemas que han surgido.

6.1.1 Posición y movimiento de unidades

Para explicar el tratamiento de mapas en *StarCraft* es necesario explicar antes varios conceptos.

La anchura y altura de los mapas puede ser de 64x64 128x128 o 256x256. Cada **casilla lógica de 1x1**, a su vez se subdivide en **32x32 posiciones físicas**. Estas posiciones físicas son los pixeles del juego (figura 52). Por lo que, al codificar los estados utilizamos solo las casillas lógicas, ya que si utilizáramos las posiciones físicas, incrementaría el tamaño de la tabla de estados en 32x32.

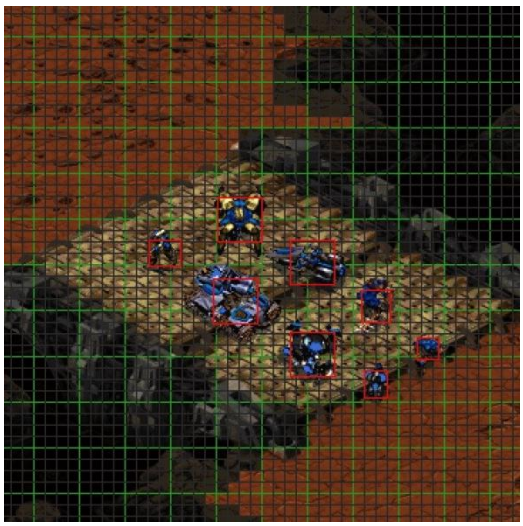


Figura 52. Posición física de las unidades en *StarCraft*

Esto provocó una serie de problemas que describiremos a continuación.

La ubicación de los distintos tipos de unidades (edificios, *add-ons*, tropas, etc.) en *StarCraft* se representa mediante una posición física, que corresponde con el centro de la unidad, y unas dimensiones de altura y anchura (figura 53).

Asimismo, en la figura 54, podemos observar como los edificios tienen una representación lógica (zona coloreada en verde), que es usada por el juego para impedir al jugador construir un edificio sobre otro edificio o unidad (figura 54). Pero en realidad el espacio real es menor (está representado por el recuadro blanco) [31] .

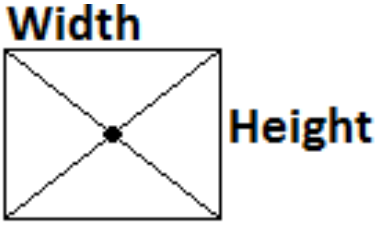


Figura 53. Ancho y alto de las unidades en *StarCraft*

Figura 54. Representación lógica y representación física de unidades en *StarCraft*

Con el uso de unidades lógicas se simplifica la representación de las unidades y edificios en el mapa, pero se crean incoherencias con el mundo físico. El problema surge cuando juntamos varios edificios, y un edificio no abarca todo el espacio físico que su representación lógica dice que ocupa. Las unidades soldado podrían atravesar el espacio existente entre varios edificios originando problemas en el algoritmo, ya que como mencionamos al principio del capítulo, codificamos los diferentes estados igualándolos a las casillas lógicas del *StarCraft*. Así pues, toda la representación lógica del edificio se declara como terreno por el cual la unidad soldado no puede pasar, cuando en realidad sí puede pasar.

Para evitar estos problemas decidimos mover las unidades del centro exacto de una casilla lógica al centro lógico del siguiente.

6.1.2 Diseño de los mapas

Como se observa en la Figura 55, los mapas de *StarCraft* que se acabaron creando, y sobre los que se realizaron los experimentos, son semejantes a los mapas del laberinto.

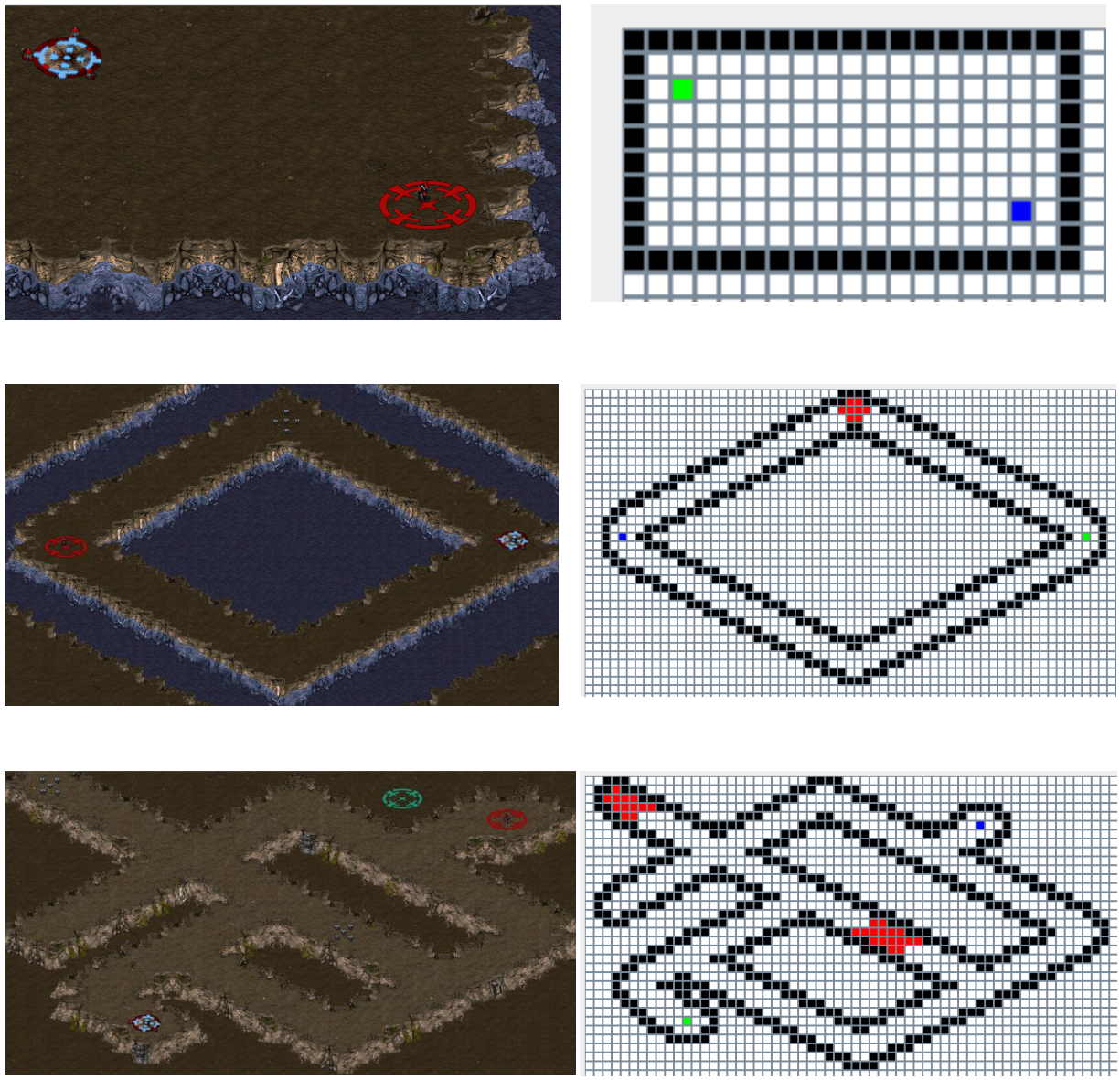


Figura 55. Mapas lógicos frente a mapas de *StarCraft*

Sobre dichos mapas cabe destacar varias cosas:

- Carecen de localizaciones: para localizar la meta en el mapa hemos decidido tomar las unidades "*Beacon*" del juego como metas.

- Las casillas que en el laberinto supondrían la muerte se han sustituido por Minas Araña Terran. Estas no matan instantáneamente, pero una vez se entra en su rango de visión, te persiguen y te acaban matando.

6.2 Experimentación

Para la realización de experimentos se utilizaron dos funciones propias del *BWAPI* de la clase *Game* que permiten realizar las pruebas mucho más rápido.

- *game.setLocalSpeed(0);* que ejecuta el juego a la velocidad máxima posible.
- *game.setGUI(false);* que ejecuta el juego sin cargar la interfaz gráfica.

Mediante algunas pruebas experimentales (figura 56), pudimos comprobar que la reducción de tiempo entre tener activada la interfaz gráfica o no tenerla (con *LocalSpeed* a 0), es de casi la mitad.

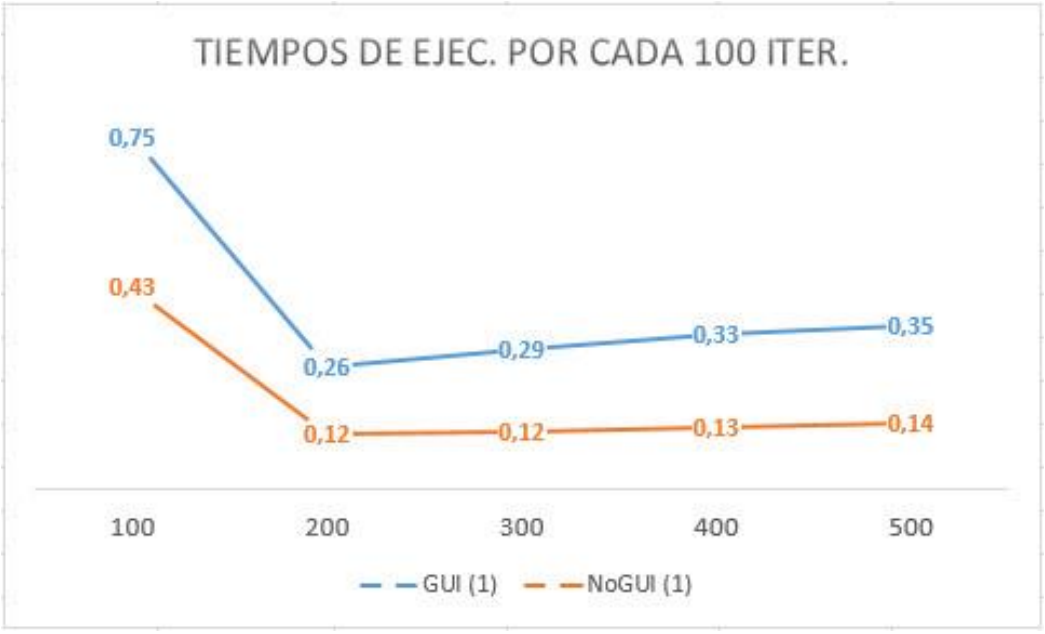


Figura 56. Diferencias entre ejecución con *GUI* y sin *GUI*

Aun con el uso de estas funciones, los experimentos sobre el *StarCraft* tenían una duración desorbitada en nuestros ordenadores (ver anexo: Especificaciones de los sistemas usados). Como muestra, la ejecución de todas las estrategias desarrolladas sobre **un único mapa** con 1000 iteraciones y 10 repeticiones, en el mejor de nuestros ordenadores tardó cerca de 10 horas. Se tuvo que reducir el número de iteraciones en el paso del mapa lógico a *StarCraft* ya que si no los experimentos resultaban imposibles.

6.2.1 Gráficas y Resultados

Habiendo realizado los mismos experimentos en ambos dominios (laberinto lógico y *StarCraft*), y habiendo modelado mapas similares, podemos comparar ambos resultados de aprendizaje y ver si coinciden los resultados entre uno y otro.

En las siguientes gráficas vamos a mostrar los resultados del aprendizaje con las 2 estrategias que han sido más representativas (estrategia básica y distancia euclídea) en cada uno de los 3 mapas (fácil, medio y difícil).

Cabe mencionar que dado a que los experimentos en *StarCraft* llevan bastante más tiempo que los experimentos con el laberinto lógico, no se ha podido realizar con el mismo número de repeticiones. Estas repeticiones servían para hallar la media de los resultados, y obtener unos resultados más fiables. Es por esto que, en los resultados en *StarCraft* la gráfica es más variable. A pesar de ello, se puede apreciar claramente la tendencia de las distintas gráficas, que es lo que vamos a valorar.

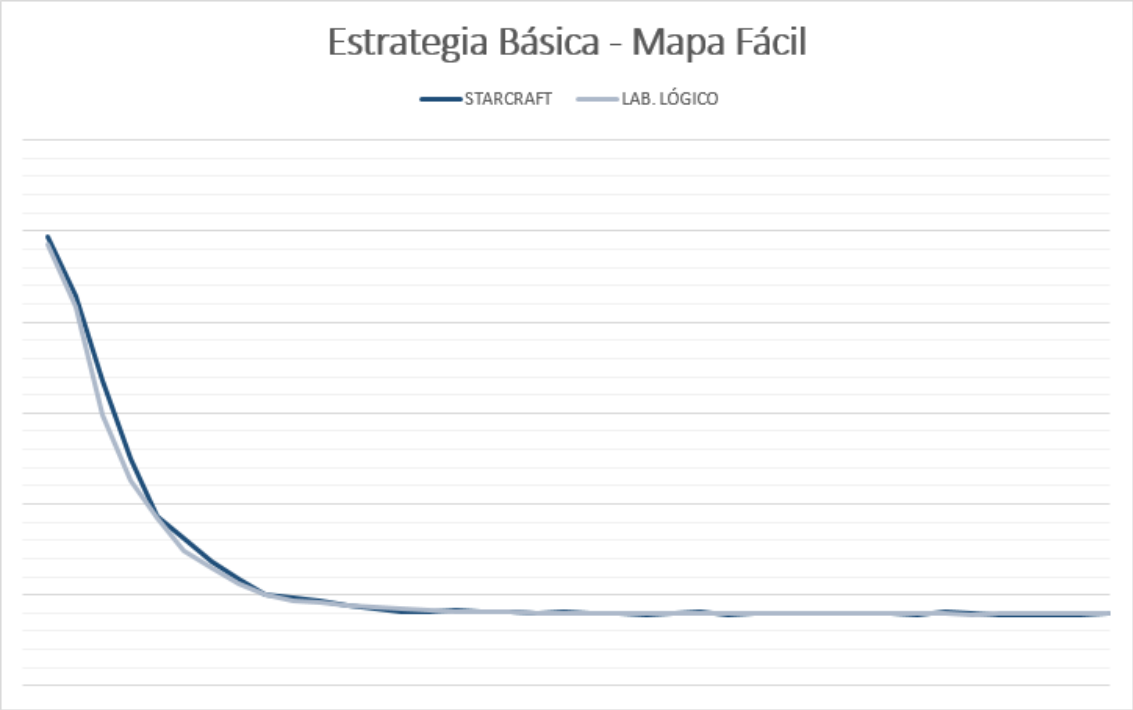


Figura 57. Comparativa entre mapa lógico y *StarCraft;* mapa fácil, estrategia básica

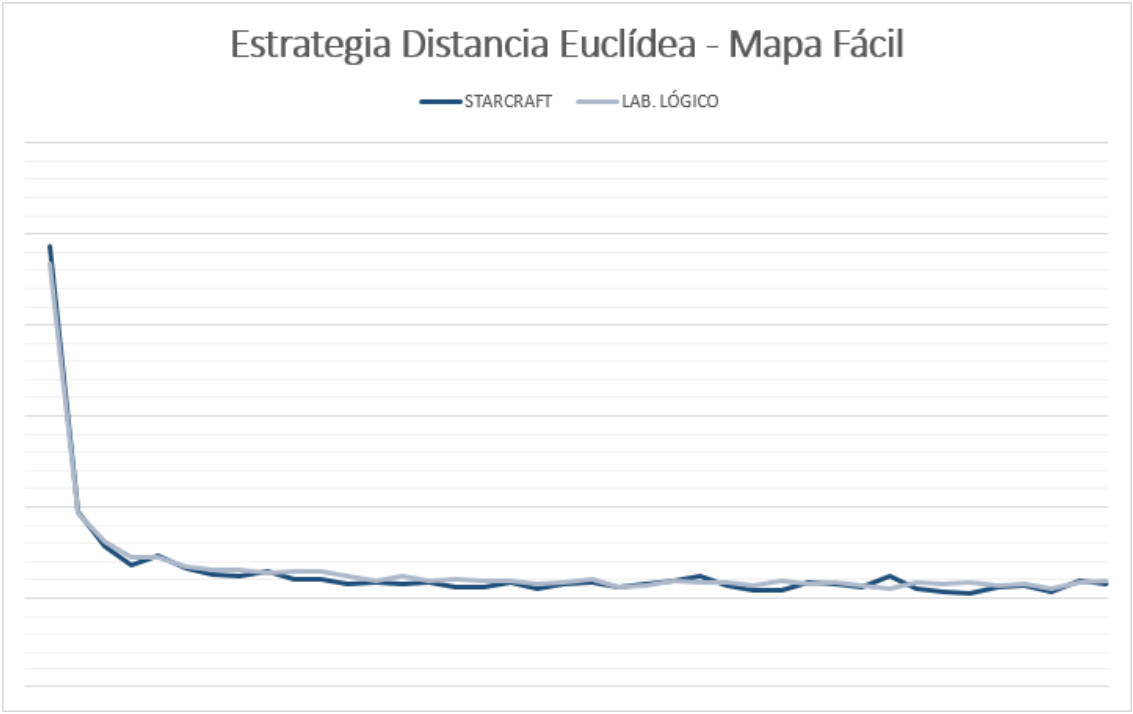


Figura 58. Comparativa entre mapa lógico y *StarCraft;* mapa fácil, estrategia euclídea

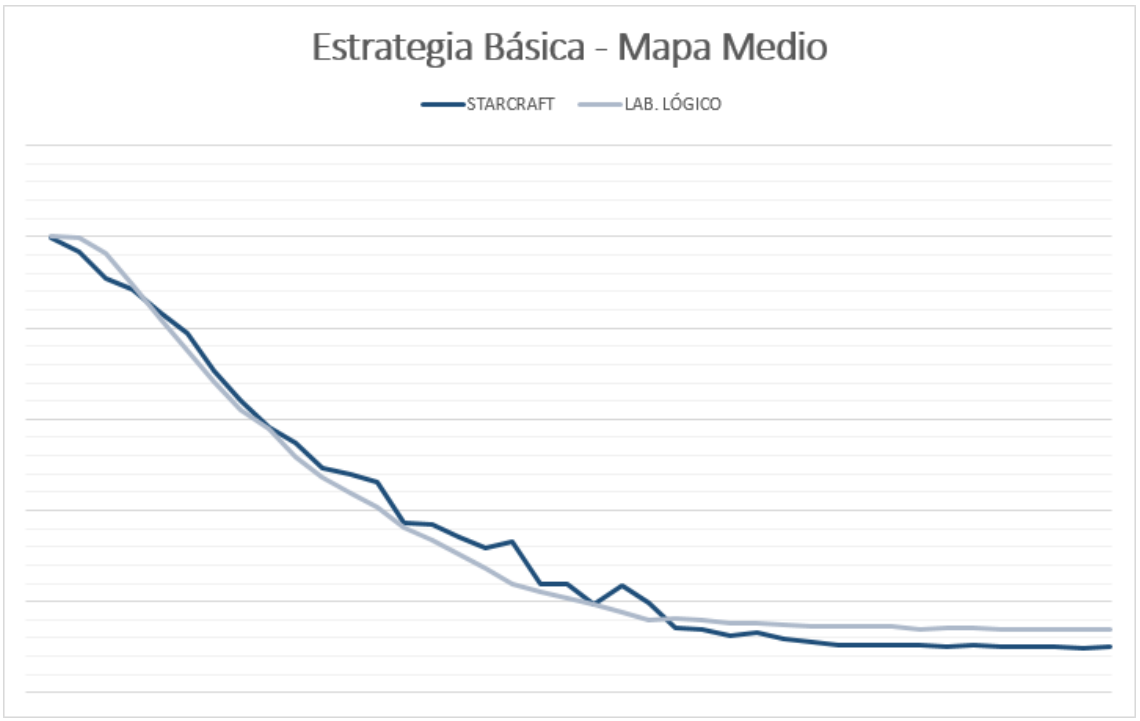


Figura 59. Comparativa entre mapa lógico y *StarCraft;* mapa medio, estrategia básica

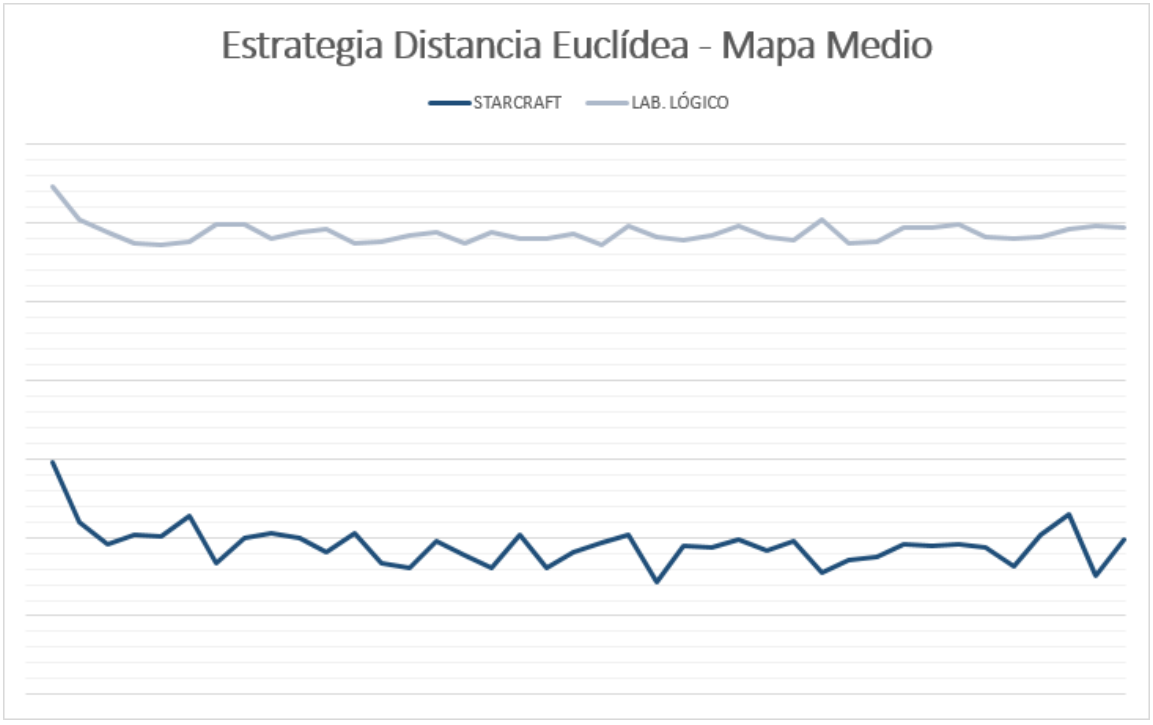


Figura 60. Comparativa entre mapa lógico y *StarCraft;* mapa medio, estrategia euclídea

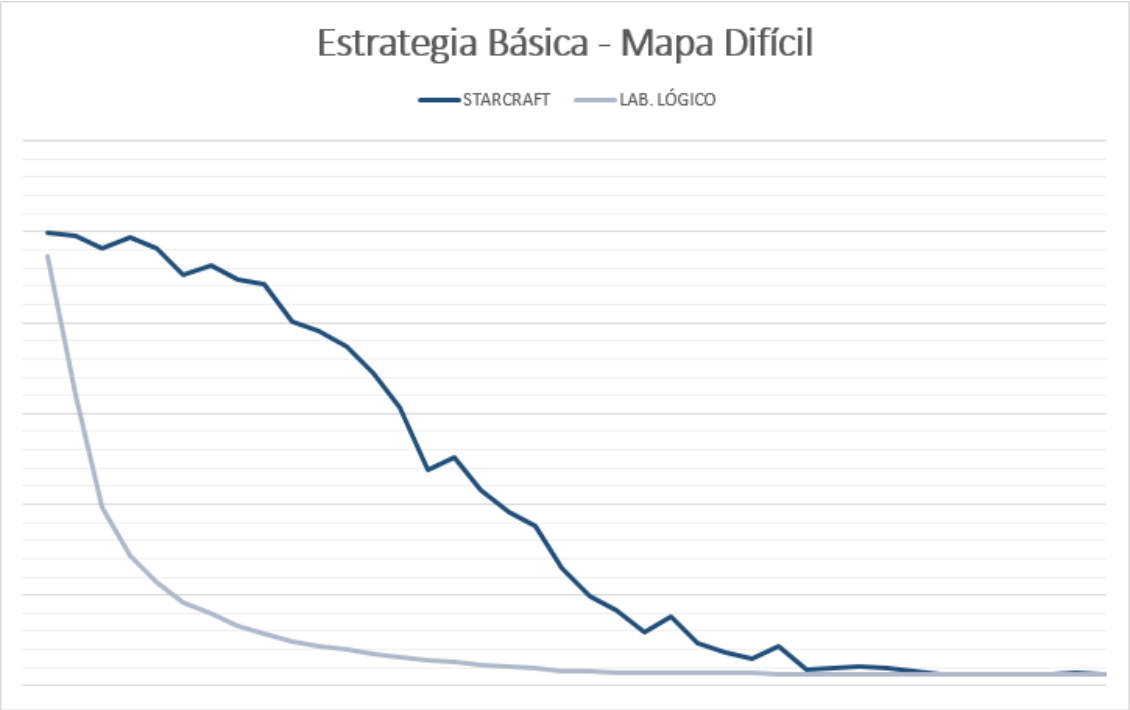


Figura 61. Comparativa entre mapa lógico y *StarCraft;* mapa difícil, estrategia básica

6.3 Conclusiones

Para resumir los resultados obtenidos, podemos decir que el mapa fácil se ha comportado de manera muy similar en ambos dominios y estrategias, por la simplicidad del mapa.

En el mapa medio hay algunas diferencias, porque mientras la estrategia básica sí que se ha comportado de manera similar en ambos dominios, con la estrategia de la distancia euclídea se ven algunas diferencias de tendencia, siendo los resultados en *StarCraft* notablemente mejores que en el laberinto lógico.

En el mapa difícil, se ve que ambos dominios muestran unas gráficas de aprendizaje bastante distintas, aunque con valores mínimos similares. Esto se puede deber otra vez a la complejidad de modelar el mismo mapa en ambos dominios, y a las pequeñas

diferencias en el comportamiento de las unidades en *StarCraft* (menos controlable) en comparación con el laberinto lógico.

Una de las hipótesis barajadas para las variaciones en el mapa medio y en el mapa difícil con respecto a los mapas lógicos es la presencia de minas, que tienen un comportamiento menos controlado. Mientras que en el laberinto lógico entrar en una casilla (estado) supone la muerte de la unidad en el *StarCraft*, la mina tiene un rango de visión amplio (figura 62), y basta con acercarse lo suficiente como para activar la mina y que te persiga. Este comportamiento puede ocasionar un aprendizaje erróneo. Por ejemplo, la unidad puede acercarse a la mina lo justo para entrar en su rango, activarla y que te empiece a perseguir. Y a partir de ese momento, la unidad empieza a ir hacia la meta. En el momento que la mina mata a la unidad, que es inevitable, el algoritmo aprenderá que esa última acción en ese estado era mala, cuando en realidad estaba yendo por el camino correcto.

Figura 62. Rango de visión de las minas en *StarCraft*

En este caso, el marine debería aprender “antes” a no acercarse hacia la zona superior del mapa.

Esta hipótesis se ve reforzada con el mapa difícil, ya que se puede observar como el aprendizaje es mucho más lento que en el mapa lógico.

Figura 63. Mapa difícil con puntos de decisión y rango de las minas

En este mapa, aparecen más puntos de decisión (puntos en blanco de la figura 63). En estos puntos una mala decisión puede implicar su muerte, debido a que también existen minas, que podrían inducir a un aprendizaje incorrecto por lo que hemos comentado antes.

Por último podemos decir que los resultados varían más entre en un dominio u otro cuanto más complejo sea el mapa que queramos modelar, y son más fiables con la estrategia básica. De todas formas, a pesar de que haya algunas diferencias entre los resultados de ambos modelos, sí que puede ser interesante emplear un modelo lógico para realizar pruebas que lleven mucho tiempo de proceso. Con estas pruebas se pueden establecer resultados generales, y con estos resultados ya ejecutar las pruebas que parece que vayan a funcionar mejor sobre *StarCraft*.

Capítulo 7. Conclusiones y trabajo futuro

7.1 Conclusiones

El proyecto que empezamos a desarrollar a principio del curso ha evolucionado con nosotros. Tuvimos que modificar algunos de los objetivos que nos planteamos al comienzo del curso, pero hemos podido completarlos satisfactoriamente. Nuestro proyecto es capaz por sí solo de aprender un camino a la salida de un laberinto. Y como hemos demostrado, no sólo en un entorno lógico, sino también dentro del entorno de *StarCraft*.

Podemos enmarcar los siguientes objetivos como cumplidos:

- Aprender a usar *Q-Learning* e implementarlo de forma eficiente dentro de nuestro proyecto. Hemos comprendido su funcionamiento, y encontrado experimentalmente unos valores de *alpha* y *gamma* bastante eficientes para nuestro problema. Así como que la estrategia de recompensa más efectiva es aquella que solo da una recompensa al llegar al final.

- Hemos aprendido a manipular el entorno de desarrollo de *StarCraft* a través del *framework* de *BWAPI*. Hemos comprendido su funcionamiento y hemos sido capaces de ejecutar el algoritmo *Q-Learning* en él.

- Hemos desarrollado un *framework* (*Teseo Q-Learning Framework*) completamente escalable y reutilizable, con el que poder trabajar con el algoritmo *Q-Learning* sin tener la necesidad de comprender en toda su profundidad el funcionamiento del algoritmo. Este *framework* es capaz de funcionar en nuestro laberinto lógico y *StarCraft*, y escalable también a otros dominios.

- Hemos gestionado de manera correcta los problemas que nos han ido surgiendo. Por ejemplo, facilitarnos el trabajo haciendo primero pruebas con el entorno lógico.

- El coste de computacional de los experimentos ha sido muy elevado incluso para un problema sencillo como el que queríamos resolver. Es posible que para problemas más complejos el coste sea inasumible. Además será necesario emplear más información del dominio, y buscar representaciones adecuadas para cada problema. También se podrían diseñar IA a distintos niveles de abstracción que se ejecutarán según la situación: a nivel estratégico, a nivel de grupos de unidades, individuales etc.

- Relacionado con el punto anterior, encontramos el problema del espacio que ocupa la *QTabla*. En problemas complejos, el número de estados posibles crece rápidamente dando lugar a *QTabla*s que ocupan mucha más memoria. Esto indicaría la necesidad de dividir o abstraer los problemas complejos antes de aplicar el algoritmo de aprendizaje.

Para finalizar debemos mencionar que al tratarse de un campo en el que no teníamos mucha experiencia, y con el reducido espacio de tiempo del que disponíamos, no hemos podido realizar tantos avances como en un principio teníamos previstos. Con nuestro trabajo hemos sentado la base de una fructífera línea de proyectos. Además, hasta donde nosotros sabemos, se trata del primer proyecto de grado que combina el aprendizaje por refuerzo con juegos de estrategia en tiempo real en el departamento.

7.2 Trabajo futuro

Gracias a la alta modularidad del proyecto desarrollado se podrían utilizar los algoritmos de aprendizaje para aplicarlos a diferentes dominios. Por ejemplo, se podría adaptar a otros videojuegos como el *Warcraft 3*. Aunque en principio este proyecto está pensado para aplicarlo a videojuegos, se puede adaptar a otros dominios siempre y cuando se puedan representar en forma de entorno, estados y acciones.

Algunas posibles mejoras a desarrollar en torno a *StarCraft* podrían ser:

- Introducir nuevas unidades en el laberinto. Gracias al editor de mapas podríamos insertar nuevas unidades para plantear nuevas situaciones en el entorno. Estas nuevas unidades obligarían a codificar nuevos estados para la *QTabla*, lo que además nos llevaría a codificar posibles nuevas acciones, como se desarrolla en el siguiente punto.

- Codificar nuevas acciones de la unidad. Actualmente la unidad tiene pocas acciones definidas. Se podrían integrar nuevas acciones como “Mantener Posición” o “Atacar”. Dentro de ellas además se deberían definir diferentes patrones de conducta. Por ejemplo, dentro de la acción “Mantener posición” podría haber dos tipos de conducta, una “*scout*” que en cuanto detecte al enemigo huya a una posición resguardada o un que saltara a la acción de “Atacar”. La cual a su vez podría dividirse en distintas conductas de ataque conocidas dentro de los RTS (Correr y disparar *hit and run*, atacar al más cercano, al de menor vida…).

- Batallas. Introducir dos grupos de marines enfrentados, uno manejado por el ordenador y otro con nuestro *bot* y con sus nuevas acciones. Esta situación, en apariencia trivial, no lo es ya que supondría el control de varias unidades por parte del *bot*. Se podrían tomar distintas decisiones cómo tratar las unidades en grupo o de forma individual. Tratarlas en grupo podría resultar poco efectivo, sin embargo, tratarlas de manera individual podría ser muy costoso, ya que como se ha podido observar durante

el desarrollo del proyecto el aprendizaje y las pruebas llevan mucho tiempo. También se podría recompensar a las unidades según el tiempo que permanezcan vivas, lo cual en entornos grandes podría resultar en el desarrollo de patrones de huida.

- Objetivos secundarios. Son objetivos que no son necesarios para conseguir completar la misión pero que ayudarán al jugador (en este caso nuestra IA) a conseguir su objetivo principal de manera más sencilla. Estas sub-metas deberían traducirse en nuevos estados que podrían implementarse con una alta recompensa de tal forma que la IA aprendiera que consiguiendo el objetivo secundario el objetivo primario sería más fácil.

- Construcción y recolección. Construir edificios y recolectar recursos requeriría un tratamiento mucho más específico del entorno. Los edificios tienen muchas variantes según su raza, la más importante aparte de obviamente su utilización, coste y espacio, es la forma de construirlos según cada clase. Según cada raza se deberían implementar acciones específicas que controlarán los trabajadores para construir el edificio mientras se continúa recolectando los recursos para seguir construyendo más edificios.

- En unión con el punto anterior también sería necesario codificar en el entorno el árbol de tecnologías de cada raza de tal forma que aprendiera que para conseguir construir ciertos edificios es necesario anteriormente construir otros.

Dentro de las líneas futuras del *framework* para *Q-Learning* podríamos destacar:

- Permitir la adopción de distintas políticas de selección de acciones. Ahora sólo se tiene en consideración una política meramente probabilística; se podrían implementar otras políticas como *greedy*, *ε-greedy*, *soft-max*…

- Mejora del algoritmo de *Q-Learning* para considerar el uso de valores negativos en la *QTabla*, y la posible retro propagación de las recompensas negativas [29] .

- Mayor abstracción del *framework*, para aceptar otros algoritmos de aprendizaje por refuerzo como puede ser *SARSA*, empleado en entornos no deterministas [32].

- Mejorar la implementación de la clase *StarCraftEnvironment*. Ya que al aumentar la complejidad del entorno crece la complejidad para codificar estados y acciones, algunos métodos de la interfaz tendrán que ser modificados para acomodarse al nuevo entorno [33]. Por otro lado, ya no valdrá definir que se ha cambiado de estado cada vez que la unidad se encuentra parada, ya que al aumentar la complejidad de cada estado igual conviene emplear otros métodos para definir cuándo se debe ejecutar una nueva iteración del proceso de aprendizaje.

Como se puede observar, existen un gran número de posibilidades para continuar con este proyecto. El propósito a largo plazo sería lograr construir una IA con distintos módulos, cada uno encargado de la realización de cierta tarea, capaces de aprender por sí mismos y de ofrecer buenos resultados tras un número determinado de ejecuciones. De esta forma lograríamos un *bot* altamente efectivo en *StarCraft*. Y quién sabe si en un futuro se podrá competir en alguno de los torneos internacionales.

Capítulo 8. Conclusions and future work

8.1 Conclusions

The project has evolved as we worked on it. We had to modify some goals we set initially, but we successfully achieved. Our project is able to learn a way out of a maze by itself, and as we have demonstrated, not only in a logical environment but also within the environment of *StarCraft*.

We can mark the following goals as accomplished:

- *Learning* to use *Q-Learning* and implement it efficiently within our project. We have understood how it works, and found experimentally an efficient *alpha* and *gamma* values, that are appropriate to our project. So, the most effective reward strategy is the one that only gives a reward when reaching the end.

- We have learned how to manipulate the *StarCraft* developing environment through the *BWAPI framework*. We fathom its operation and we were able to run the *Q-Learning* algorithm in the project.

- We have developed a fully scalable and reusable *framework* (*Teseo Q-Learning Framework*); with this *framework* it's possible to work with the *Q-Learning* algorithm without needing to comprehend in all of its depth the algorithm operation. This *framework* is capable of operating in our logical and *StarCraft* maze, and it's also scalable to other domains.

- We have managed to solve the problems that we have faced. For example, easing our work by testing first the logical environment.

- The computational cost of the experiments has been very high even for a simple problem as the one we wanted to solve. For more complex problems maybe the cost

would be unacceptable. Also, it will be necessary to use more domain information, and to find more efficient models for each problem. IA at different levels of abstraction could also be designed to be implemented according to the situation: at strategic level, at level of groups of units, individual, etc.

- Related to the previous point, we face the problem of space the *QTable* needs. In complex problems, the number of possible states grow up quickly generating large *QTable*s in memory. On our project it has a remarkable size but handling. If states and actions changed their representation to become more complex, the *QTable* would quickly grow to a size hardly treatable. This would indicate the need to divide or abstract the complex issues before applying the learning algorithm.

Finally mention that, as it is a field in which we didn't have much experience, and with the reduced space of time available, we haven’t been able to make all the progress we expected. With our work we have laid the basis for a fruitful line of projects. Moreover, to our knowledge, this is the first grade project which combines reinforcement learning with strategy games in real time in the department.

8.2 Future work.

Thanks to the high modularity of the project developed learning algorithms could be applied on different fields. For example, it could be adapted to new games such as *Warcraft 3*. Although at first the project is intended to be applied on video games, it can be adapted to others, provided that the domain can be represented as environment, states and actions.

Possible improvements to develop around *StarCraft* could be:

• Introducing new game-units in the maze. Thanks to the map editor these new units could be added to reach new situations in the environment. These new units would force to encode new states in the *QTable*, which also leads us to encode possible new actions, as developed in the following section.

• Encoding new actions of the game-unit. Currently the game-unit has few actions defined. New actions could be integrated as "Hold Position" or "Attack". Among them different patterns should also be defined. For example, within the action "Hold Position" could be two types of behavior: a "scout" that when it detects an enemy, it flees to a sheltered position or other that would jump to the action of "Attack", which in turn could be divided into different attack behaviors known within the RTS (Run & Shoot hit and run, attack the nearest enemy first, the life lowest ...).

• Battles. Insert two groups of marines, one managed by the computer and the other with our IA and our new actions. This situation, seemingly trivial, it's not because the *bot* would control several units. Different decisions could be taken like treating the game-units as a group or individually. Treating them as a group could be ineffective; however, treating them individually could be very expensive, because as it's been observed during the development of the project, learning and tests would take a long time. The units could also be rewarded according to the time they remain alive, which in large environments could result in the development of runaway patterns.

• Secondary goals. These are goals that aren't necessary to achieve complete the mission but which may help the player (in this case our IA) to achieve its main goal more easily. These sub-goals should translate into new states that could be implemented with a high reward so the AI would learn that by getting the secondary target, the primary objective would be easier to reach.

• Construction and gathering. Constructing buildings and gathering resources would require a more specific treatment of the environment. Buildings have many variations depending on the breed, being the most important, apart from obviously its use, cost and space, how to build them according to each class. That happen because each race should implement specific actions that will control workers to construct the buildings while at the same time the IA must continuing gathering resources to continue to build more buildings.

• Consequently with the previous point, it would also be necessary to codify the environment of technology tree of each race such a way they learn that for getting built certain buildings previously is needed to build others.

Within the *framework* for future lines of *Q-Learning* we could highlight:

• Allowing the incorporation of different selection policies actions. Now it only takes into account a purely probabilistic policy; other policies could be implemented as greedy, ε-greedy, soft-max...

• Changing the "*alpha*" and "*gamma*" values at runtime, so they could adapt to changing performance and the importance of rewards as execution proceeds.

• Improved *Q-Learning* algorithm to consider using negative values in the *QTable*, and the possible propagation of negative rewards [29] .

• Increased *framework* abstraction, to accept other reinforcement learning algorithms such as SARSA [32].

• Improve implementation of the *StarCraft* Environment class. Since increasing complexity of the environment becomes more complex to encode states and actions,

some of the interface methods should be modified to suit the new environment. Worth no longer define what has changed every time the unit is shutdown, since increasing the complexity of each state should just use other methods to define when to run a new iteration of the learning process. One strategy might be to run iteration from time to time.

As show, there are a large number of possibilities to continue this project. The long-term aim would be to build an AI with different modules, each one responsible for carrying out certain tasks, able to learn by themselves and to provide good results after a certain number of executions. In this way we would achieve a highly effective *bot* in *StarCraft*. And who knows if in the future will be able to compete in any international tournaments.

Capítulo 9. Aportaciones individuales al proyecto.

En este capítulo vamos a hacer un pequeño recorrido por las distintas fases del desarrollo, explicando las contribuciones individuales que cada componente del grupo ha aportado al proyecto.

9.1 Juan Deltell Mendicute

Mis aportaciones a este trabajo han estado distribuidas por todo el proyecto. Empezando por las pruebas y *testing* de la interfaz de *BWAPI* para Java. Empecé haciendo pequeñas líneas de código para ver cómo funcionaban los comandos y cómo respondían a esas acciones. El objetivo era ir investigando su completo funcionamiento, y al final creé un módulo de IA que fue capaz de resistir en una partida los primeros dos ataques que te hacía la máquina. Con esto conseguí las órdenes necesarias para mover a un marine espacial en es *StarCraft* entre otras cosas.

Debido a que tengo el ordenador más potente del grupo (ver anexo), se me encargó a mí la realización de gran parte de las pruebas y la recolección de datos de las mismas. He de recordar que estas pruebas para un único mapa llegaban a tardar más de diez horas. Pese a que mi ordenador es bastante moderno, no llega a dos años de antigüedad, y es de gran potencia, no se pudo hacer la última prueba de con la estrategia euclídea en el mapa difícil. El problema era el gran uso de memoria RAM que usaba el compilador java, Se intentó aumentar el tamaño del *heap* de la máquina virtual y reducir el número de iteraciones considerablemente, pero seguía dando el mismo problema, así que se dejó sin hacer. Con las pocas pruebas que se consiguieron sacar sin este problema, se pudo verificar que también seguía la misma pauta que en el mapa lógico.

Con respecto al código estuve haciendo alguna de las posibles estrategias, para resolver nuestro problema. Entre ellas la estrategia de *"less steps", "repeated states"* y la estrategia que juntaba todas las estrategias anteriores. Con la estrategia *"repeated states"*, el algoritmo llegaba a un punto que se quedaba totalmente bloqueado. Sobre todo si el agente se topaba con un obstáculo o con el muro del laberinto. Donde las opciones se reducían bastante, llegándose a bloquear si llegaba a una posición donde todas las posiciones de su alrededor habían sido ya visitadas. Esta estrategia en concreto se descartó rápidamente, aunque pudimos haberle dado alguna vuelta de tuerca más, porque en la teoría resultaba ser muy buena. Como ya teníamos suficientes estrategias la descartamos.

En todas estas estrategias también tuve que hacer sus correspondientes pruebas para su correcto funcionamiento y posibles mejoras de éstas para un mejor rendimiento. Al final, se pudo observar que la mejor estrategia fue la estrategia básica, con una única recompensa para cumplir su objetivo. O sea, que estas estrategias fueron ineficientes desde el punto de vista práctico. Para que fuesen más eficientes habría que ir modificando dinámicamente los valores de *alpha* y *gamma*, porque nos son igual de buenos para todas las estrategias. Aparte, tendríamos que depurar más todas estas estrategias para encontrar su punto de optimización.

Para la parte de la memoria, empecé haciendo la estructura del documento. Su división en los capítulos pertinentes y con un formato correcto.Busqué información sobre aprendizaje por refuerzo, aprendizaje supervisado y no supervisado y *Q-Learning*, y empecé a hacer una primera versión del resumen, motivación y el apartado de aprendizaje por refuerzo y *StarCraft*. Todo esto se fue modificando conforme íbamos avanzando con el proyecto, hasta alcanzar el estado actual. También me ocupé del índice, las figuras y su índice, y de introducir la bibliografía.

También hice el apartado 5.3, donde se explican las diferentes estrategias realizadas en el proyecto, los resultados de éstas, así como las conclusiones que sacamos de haber utilizado estas estrategias, y la explicación de por qué algunas estrategias fueron buenas y otras fueron malas.

También contribuí en la elaboración de las conclusiones finales explicando los resultados obtenidos. Por último, puse unas posibles vías de futuros trabajos que se pueden hacer a partir de ese proyecto.

9.2 Alberto Lorente Sánchez

Mis aportaciones personales al proyecto han sido de lo más variadas.

Al comienzo del desarrollo creé varios mapas de *StarCraft* tanto para aprender a usar el editor como para realizar una toma de contacto con la nueva tecnología a través de la librería *BWMirror*.

En el editor de *StarCraft* aprendimos a crear mapas utilizando las *settings* propias del mismo para crear el dominio de aprendizaje deseado. Dentro de dichas *settings* encontramos los *triggers*. Estos *triggers*, tienen el mismo funcionamiento que los disparadores de las bases de datos, cuando se cumple una condición se ejecuta una consecuencia. Para poder jugar los mapas como nosotros deseábamos era necesario eliminar los *triggers* predefinidos que establecían las condiciones necesarias para ganar la partida y terminar el juego.

Inicialmente queríamos desplazar el marine hacia un “área” del juego (en el editor de mapas se pueden crear estas secciones de mapa concretas), utilizándola como meta para lo que utilizamos los *triggers* anteriormente mencionados. Cuando el jugador desplazaba la unidad hasta el “área” meta en el mapa el *trigger* cumplía la condición y marcaba el final del juego con victoria. Esta idea se mantuvo durante un poco menos de un mes y permitió comprobar el buen funcionamiento del módulo de aprendizaje. Sin embargo, fue finalmente descartada por dos motivos: la imposibilidad de depurar los errores de código y la generación de “ruido” en los resultados. Este ruido se provocaba porque los *triggers* del mapa se comprueban cado 2 segundos *ingame* y provocaba que el marine pudiera moverse antes del evento victoria. En vez de un “área” de salida se tomó la “*Baliza Terran*” como marca de salida tal y como está actualmente.

Sobre la librería *BWMirror* ya en el código fuente aporté varias secciones remarcables.

- El tratamiento a nivel físico del marine. Sus movimientos, controlando en todo momento que el movimiento que iba a realizar era posible, es decir, que cumplía todos los requisitos necesarios para moverse. Estos requisitos implican: no salirse del mapa, poder trasladarse del centro de una casilla lógica al centro de la siguiente (si existe un camino), y sobre todo que no colisionará con ninguna unidad en la realización del movimiento (ya que esto provocaría errores en la codificación del dominio). Este punto es especialmente importante ya que como se explica en el capítulo 6.1.1 “Posición y movimiento de unidades” de *StarCraft* las unidades no tienen la misma codificación física que lógica.

- Tratamiento de datos mediante el uso de las librerías de Excel. Como se puede observar todos los gráficos expuestos en este documento están realizados con Microsoft Excel por su gran simplicidad y amplia posibilidad de gráficos. Al comienzo del proyecto extraíamos los datos de la aplicación en texto plano para luego copiarlos al Excel. Esto provocaba una pérdida de tiempo notable. Además para los gráficos expuestos en el punto 5.3 “Experimentos con estrategias de recompensa” de esta memoria era necesario almacenar en varios ficheros de texto el número de pasos recorridos por las casillas lógicas. Una vez añadida y bien implementada la librería de Excel, todo esto pasó a ser automático siendo solo necesario analizar los datos una vez escritos en el propio fichero.

- Creación e implementación de estrategias en el laberinto lógico. Al implementar el *Environment* del laberinto lógico se propusieron varias estrategias que mejoraran el aprendizaje en base a modificar las funciones de recompensa. Estas estrategias pueden son ampliamente explicadas en el punto 5.3 “Experimentos con estrategias de recompensa” de esta memoria. En total se crearon 5 estrategias con el fin de mejorar los resultados de aprendizaje. Cabe mencionar que algunas fallaron porque utilizamos unos valores de alpha y gamma constantes. También hay que añadir que fueron los primeros experimentos de gran volumen que se realizaron con el módulo de aprendizaje. Estas pruebas, aun siendo en el laberinto lógico, supusieron una gran inversión de tiempo.

Centrados en la realización de este mismo documento me corresponde la creación de varios apartados. Estos apartados serian la explicación del funcionamiento de *Q-Learning* así como de alpha y gamma. También le apartado de *StarCraft* y de sus

conclusiones. Además de una considerable aportación al último apartado tanto en conclusiones finales como en líneas futuras de trabajo. Por último al igual que mis compañeros revisamos el documento al completo varias veces realizando correcciones sobre el mismo.

9.3 Jesús Martínez Dotor

Mis aportaciones al proyecto se han centrado sobre todo en el algoritmo de aprendizaje *Q-Learning*, el desarrollo del *framework Q-Learning*, y la implementación e integración de este en los dos dominios planteados (laberinto lógico y *StarCraft*) para la realización de las distintas pruebas. Con todo ello, también he realizado algunos aportes en torno a temas exclusivos de *StarCraft* y la librería *BWMirror*. En las siguientes líneas intentaré describir más en detalle estas aportaciones.

En cuanto a *BWMirror*, la primera aportación destacable fue la generación de pruebas en torno a la velocidad de ejecución. Los resultados, que se pueden ver en el apartado “6.2 Experimentación”, demostraron que la reducción de tiempo al desactivar la interfaz gráfica del juego era de aproximadamente la mitad. Además, estas primeras pruebas sirvieron para generar una estructura que más tarde reutilizaríamos en las pruebas de los parámetros de *Q-Learning* y estrategias de aprendizaje.

Una segunda parte sin relación directa con el algoritmo *Q-Learning* fue la reutilización de una práctica de Ingeniería del Conocimiento para la creación del dominio de laberinto lógico y la implementación de la herramienta de creación de laberintos, comentada en el apartado “4.3 Herramientas de diseño de los experimentos y análisis de datos”. Esta parte no me llevó mucho tiempo, pues al reutilizar un código anterior sólo había que descartar las partes que no usábamos (relacionadas con el algoritmo *A**) e implementar las nuevas funcionalidades: posibilidad de añadir trampas, y carga/salvado de mapas en ficheros de texto.

Centrándonos en la parte de aprendizaje automático, desde el principio del TFG dediqué una gran cantidad de tiempo a estudiar el funcionamiento del algoritmo *Q-Learning*, saber

dónde se encuadra dentro del aprendizaje por refuerzo, y conocer sus principios básicos y las diferencias con otros algoritmos similares. Este estudio me sirvió para desarrollar una primera versión funcional del algoritmo sobre *StarCraft*, codificada directamente sobre *BWMirror*. Además, con esta primera versión pudimos comprobar experimentalmente cómo influían en el aprendizaje los valores *alpha* y *gamma*, además de la función de evaluación *Q(s,a)*. También sirvió para detectar algunos problemas derivados de la incorrecta implementación del algoritmo; como fueron la incorrecta actualización de la *QTabla*, y algunos errores con las recompensas o la política de selección de acciones.

Después de un par de meses trabajando con el código del algoritmo junto al código de *StarCraft* cada modificación sobre el mismo se hacía más compleja y la depuración era más complicada, por ello decidimos que lo más práctico sería dividirlo en dos partes: *Q-Learning* y *StarCraft*. Así es como surgió de forma natural la idea del *framework* para *Q-Learning*; por la necesidad de mejorar el fácil mantenimiento y escalabilidad del código. El proceso de diseño del *framework*, explicado en el capítulo “4.2 *Teseo Q-Learning Framework*”, me llevó varias semanas, y fue necesario un estudio más profundo del algoritmo, pues era necesario diseñarlo de tal forma que fuera lo suficientemente abstracto como para aceptar todas las posibles variaciones del algoritmo, y fuera fácilmente trasladable de un dominio a otro. El objetivo era que una vez implementado, se pudiera continuar con el desarrollo del proyecto sin tener que modificar el código del *framework* y sin preocuparnos en profundidad de la parte de *Q-Learning*. Esto se cumplió sólo en parte, ya que una vez integrado con el dominio del laberinto lógico, tuve que realizar algunas pequeñas modificaciones que se escaparon del diseño inicial. Después de implementar el *QLearner*, se realizó además la implementación de una versión modificada que llamamos *QPlayer*, que podía funcionar siguiendo la política almacenada en una *QTabla*, obtenida en un proceso de aprendizaje anterior.

Una vez desarrollado el *Teseo Q-Learning Framework*, hubo que realizar la instanciación del mismo en los distintos dominios. En un principio se realizó la instanciación en el laberinto lógico, con el que pudimos realizar las pruebas iniciales del algoritmo, como se

explica en el capítulo “5. Resolución de laberintos lógicos”. Para la instanciación del *framework* sobre *StarCraft* fui integrando el trabajo que hacían mis compañeros con el movimiento de unidades y reconocimiento de estados en *BWMirror*.

Además, planteamos los tres niveles que utilizamos para las pruebas, tanto en el laberinto lógico como en *StarCraft*. Se modelaron intentando hacer que fueran similares en ambos dominios, y que tuvieran algún tipo de reto que nos permitiera verificar el correcto funcionamiento del algoritmo de aprendizaje.

En cuanto a la parte de las pruebas con distintas estrategias de recompensa, me ocupé de ir verificándolas e integrándolas dentro de las clases *Environment* de cada dominio, y de implementar el código para realizar dichas pruebas de forma automática.

Una vez generadas todas las pruebas y con los datos de la velocidad de aprendizaje, obtenidos gracias al uso de la librería de Excel por Alberto, se generaron las distintas gráficas en Excel para poder visualizarlos correctamente. En este proceso fue necesario trabajar un poco más con los distintos valores, sacando la media de los resultados por tramos, para suavizar la curva de las funciones de aprendizaje.

Por último, dentro de la memoria me encargué principalmente de redactar el capítulo 4, en el que se habla de la arquitectura de nuestro sistema y del *framework* para *Q-Learning*, además de elaborar los distintos diagramas de clases y de secuencia. También colaboré con el capítulo 3, que desarrolla en profundidad el tema de aprendizaje por refuerzo y *Q-Learning*, y generé las gráficas de las distintas pruebas del algoritmo y contribuí a encontrar las explicaciones del capítulo 5. Finalmente, aporté mi parte de conclusiones y líneas futuras, e hice la primera revisión del documento realizando, en menor medida, aportaciones a todos los apartados.

Bibliografía y referencias

[1] «Creative Commons».

[2] «Build software better, together», *GitHub*. [En línea]. Disponible en: https://github.com. [Accedido: 15-jun-2015].

[3] «gnu.org». [En línea]. Disponible en: http://www.gnu.org/licenses/quick-guide-gplv3.html. [Accedido: 15-jun-2015].

[4] E. Gramajo, R. García-Martínez, B. Rossi, E. Claverie, P. Britos, y A. Totongi, «Una Visión Global del Aprendizaje Automático», *Rev. Inst. Tecnológico B. Aires*, Volumen 22, p. Páginas 67–75, 1999.

[5] «Trello». [En línea]. Disponible en: https://trello.com. [Accedido: 15-jun-2015].

[6] Linus Torvalds, *Git*. 2015.

[7] *Google Docs*. Google, 2015.

[8] «Google Drive: almacenamiento en la nube, copias de seguridad de fotos, documentos y mucho más». [En línea]. Disponible en: https://www.google.com/drive/. [Accedido: 15-jun-2015].

[9] T. M. Mitchell, *Machine Learning*, 1 edition. New York: McGraw-Hill Education, 1997.

[10] R. Kohavi y F. Provost, «Glossary of terms», *Mach. Learn.*, vol. 30, pp. 271–274, 1998.

[11] T. J. Hastie, R. J. Tibshirani, y J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, 2nd. ed. corrected at 7th. print. New York: Springer, 2013.

[12] J. M. De la Cruz García y G. Pajares Martinsanz, Eds., *Aprendizaje automático: un enfoque práctico*. Madrid: Ra-Ma, 2010.

[13] R. S. Sutton, *Reinforcement learning: an introduction*. Cambridge, Massachusetts [etc.]: MIT Press, 1998.

[14] «StarCraft», *StarCraftWiki*. [En línea]. Disponible en: http://es.starcraft.wikia.com/wiki/StarCraft. [Accedido: 15-jun-2015].

[15] «The State of the RTS - IGN». [En línea]. Disponible en: http://www.ign.com/articles/2006/04/08/the-state-of-the-rts. [Accedido: 15-jun-2015].

[16] «¿Qué es Micro y Macro en StarCraft 2?». [En línea]. Disponible en: http://www.starcraft2-sc2.com/guia/que-es-micro-y-macro-en-starcraft-2. [Accedido: 15-jun-2015].

[17] «StarCraft II», *StarCraft II*. [En línea]. Disponible en: http://us.battle.net/sc2/es/game/guide/gameplay-overview. [Accedido: 15-jun-2015].

[18] «Brood War Application Programming Interface - Liquipedia Starcraft Wiki». [En línea]. Disponible en: http://wiki.teamliquid.net/starcraft/Brood_War_Application_Programming_Interface. [Accedido: 15-jun-2015].

[19] «JNIBWAPI/JNIBWAPI», *GitHub*. [En línea]. Disponible en: https://github.com/JNIBWAPI/JNIBWAPI. [Accedido: 15-jun-2015].

[20] «LICENSE - BWMirror». [En línea]. Disponible en: http://bwmirror.readthedocs.org/en/stable/LICENSE/. [Accedido: 15-jun-2015].

[21] «[SSCAIT] Student StarCraft AI Tournament 2015». [En línea]. Disponible en: http://www.sscaitournament.com/index.php?action=tutorial. [Accedido: 15-jun-2015].

[22] «aaaiaiide15». [En línea]. Disponible en: http://www.aiide.org/home. [Accedido: 15-jun-2015].

[23] «IEEE Conference on Computational Intelligence and Games | IEEE Computational Intelligence Society». .

[24] «[SSCAIT] Student StarCraft AI Tournament 2015». [En línea]. Disponible en: http://www.sscaitournament.com/. [Accedido: 15-jun-2015].

[25] Buro, «In Proceedings of the Behavior Representation in Modeling and Simulation Conference», 2004.

[26] U. Jaidee and H. Munoz-Avila, «ClassQ-Learningl: A Q-Learning algorithm for ˜ adversarial real-time strategy games», presentado en Eighth Artificial Intelligence and Interactive Digital Entertainment Conference, 2012.

[27] Maissiat, «Adaptive high-level strategy Learning in StarCraft», presentado en Proceedings of the SBGames conference on Computing, 2013.

[28] «BWMirror API - Documentation». [En línea]. Disponible en: http://bwmirror.jurenka.sk/. [Accedido: 15-jun-2015].

[29] T. Fuchida, K. T. Aung, y A. Sakuragi, «A study of Q-learning considering negative rewards», *Artif. Life Robot.*, vol. 15, n.º 3, pp. 351–354, oct. 2010.

[30] «Apache POI - the Java API for Microsoft Documents». [En línea]. Disponible en: https://poi.apache.org/. [Accedido: 15-jun-2015].

[31] «Talk:List of Unit and Building Sizes - Liquipedia Starcraft Wiki». [En línea]. Disponible en: http://wiki.teamliquid.net/starcraft/Talk:List_of_Unit_and_Building_Sizes. [Accedido: 15-jun-2015].

[32] B. D. Smart, *Reinforcement learning: A user's guide*. Washington University in St. Louis, 2005.

[33] N. Tziortziotis, K. Tziortziotis, y K. Blekas, *Play Ms. Pac-Man using an advanced reinforcement learning agent*. 2014.

Anexo

Especificaciones de los sistemas y componentes usados

Juan Deltell Mendicute

Intel Core i7-2400k 3.4GHz
16GB RAM
Windows 7 Home Premium (64 bits)

Alberto Lorente Sánchez

Intel Core i5 M430 2.27GHz
4GB RAM
Windows 7 Home Premium (64 bits)

Jesús Martínez Dotor

Intel Core i7-3610QM 2.30GHz
8GB RAM
Windows 7 Ultimate 64-bit SP1