Aprendizaje por Refuerzo
Elementos básicos y algoritmos

Lorién Lascorz Lozano

Trabajo de fin de grado en Matemáticas
Universidad de Zaragoza

Director del trabajo: José Tomás Alcalá Nalvaiz

Prólogo

El aprendizaje por refuerzo es un área del estudio informático y matemático que engloba sistemas y algoritmos de aprendizaje sobre ordenadores. En particular, esta teoría es capaz de resolver los problemas deseados sin la necesidad del conocimiento estricto y completo de todos los parámetros del problema. Esta característica es muy conveniente ya que en muchos de los problemas reales, se desconocen estos parámetros o no se controlan con exactitud. Estos sistemas de aprendizaje proporcionan al ordenador la capacidad de aprender de los datos y no sólo de ejecutar unas funciones para las que están programadas.

En este sentido, el aprendizaje por refuerzo está convirtiéndose en una herramienta muy importante en muchas áreas como por ejemplo la inteligencia artificial o la economía. Estas diversas e importantes aplicaciones son unos de los principales reclamos para el estudio de estos métodos de aprendizaje. En concreto, la aplicación de estos algoritmos a la resolución de juegos sencillos me interesó desde un primer momento. Este tipo de ejemplos nos van a ayudar a presentar los resultados a lo largo del trabajo.

El tratamiento de los problemas enmarcándolos dentro de los Procesos de Decisión de Markov hacen más asequible la comprensión del planteamiento de los métodos y algoritmos, obteniendo un marco claro y ordenado para desarrollar los problemas.

Entre todos los métodos de aprendizaje por refuerzo destaca el algoritmo *Q-learning* o Q-algoritmo. Es el método más extendido en este tipo de métodos y se encuentra implementado en el programa *R*. Estudiaremos las diferentes alternativas que nos puede brindar este programa sobre aprendizaje por refuerzo.

Abstract

Reinforcement learning is an area of computer and mathematical study that includes computer learning systems and algorithms. Particularly, this theory does not require prior knowledge of the layout of the problem in order to solve the desired problems. This feature is very convenient for many real problems, which have not enought knowledge about the problem. In this sense, reinforcement learning is becoming a very important tool in many areas such as economy or different aspects of artificial intelligence.

This procedure of learning is founded on human psychology, concerning human or animal education and behavior. It tries to imitate the didactic process based on the exercise of trial and error. In the problems that we develop in this work, the agent interacts with the environment in order to gain some information about the problem, but it only knows the final goal and when the process is over, it will receive a positive or negative reinforcement signal, depending on whether the objective has been achieved or the agent has failed.

When we have more or less information about the problem, and therefore, how much knowledge we have about the learning we want to acquire, other alternatives for learning about computers are born. We can have no information about the problem, even the final objective (*Unsupervised learning*) or, on the other hand, we can completely control the strategies we want to achieve and intermediate objectives in the problem to be developed (*Supervised learning*). All these types of learning, including reinforcement learning, are found in the same area called *Machine Learning*. The real value of these powerful tools is the ability to learn from data and not just to perform the functions which they are programmed to. At the heart of these systems are the learning algorithms, which study the data they receive, find patterns within them and define what behaviour to follow depending on the situation.

In this work, we focus on the development of reinforcement learning on problems which are structured as Markov Decision Processes. This framework fits perfectly into the learning approach, so many problems that we want to solve by this procedure are programmed according to the Markovian process scheme. In this situation, we can have no knowledge of the payoff and or transition functions, but we always have knowledge of the final objective.

First of all, we provide an introduction to reinforcement learning and machine learning. After that, we continue defining our problem space: we discuss the Markov processes and develop the main results from these processes needed for the study of reinforcement learning. We introduce the functions and parameters that we will use during the work, such as value-state function, value-action or optimal policy.

In the explanations of these concepts, we introduce the Bellman's equations and the Bellman's operator. These terms will be used in order to proof the convergence of some methods of reinforcement learning.

We then present the most important methods of applying this learning. We distinguish between two possibilities based on the information we have: model-based learning or model free learning. We present the main algorithms for each of these possibilities, including the value iteration, the policy iteration, the Q-learning algorithm and the Sarsa algorithm. We also introduce the main results with which we are able to study the convergence of these methods.

The explanation of these methods lead us to the question of exploitation and exploration, where the possibilities of control over getting new information or promoting the exercise of the information already got are analysed. This issue can be dealt from parameter values of the previous methods and from transition probabilities. In this sense, we present the possible alternatives such as the ε- greedy and

softmax method.

Next, we study the application of these methods within the language *R*, we consider some of the different libraries that *R* provides us, in particular, the *MDPtoolbox* and *ReinforcementLearning* libraries. The first one provides functions and tools for solving Markov Processes. The strategies which it follows are strategies of dynamic programming. This means that it requires all the information of the problem including actions and rewards. The second package, *ReinforcementLearning*, works as the model free of reinforcement learning itself. It makes possible to learn the best actions to maximize the long-term accumulate reward based on sequences of recorded or simulated data. We develop a simple example to explain its different tools.

Finally, we use a real game, called Wappo Game, to simulate it and obtain the optimal policies or strategies applying the Q-learning tools.

Índice general

Prólogo III

Abstract V

1. Aprendizaje automático 1
1.1. Aprendizaje por refuerzo . . . 1

2. Procesos de Decisión de Markov 3
2.1. Definiciones principales . . . 3
2.2. Desarrollo de los MDP . . . 5
2.3. Política óptima . . . 7

3. Algoritmos de aprendizaje. Q-learning 9
3.1. Aprendizaje basado en modelo . . . 9
3.1.1. Iteración de valores . . . 9
3.1.2. Iteración de políticas . . . 10
3.2. Aprendizaje sin modelo . . . 11
3.2.1. Métodos de diferencia temporal: TD(0) . . . 11
3.2.2. Q-learning: método *off-policy* . . . 12
3.2.3. Sarsa: método *on-policy* . . . 13
3.3. Exploración – Explotación . . . 14

4. El aprendizaje por refuerzo en *R* 15
4.1. Biblioteca *MDPtoolbox* . . . 15
4.2. Biblioteca *ReinforcementLearning* . . . 17
4.3. *Wappo Game*: un ejemplo elaborado . . . 19

Bibliografía 25

Anexo 1 27

Anexo 2 29

Capítulo 1

Aprendizaje automático

En la actualidad, desarrollar sistemas de aprendizaje en ordenadores es una herramienta cada vez más demandada. Los avances en este tipo de sistemas han permitido un crecimiento muy importante en el área de la inteligencia artificial. Ejemplos tan conocidos como el del juego 'Go' [10], han puesto sobre la mesa el enorme potencial de estos instrumentos con los que hoy en día ya se trabaja en grandes dimensiones y en diferentes ámbitos.

El verdadero valor de estas potentes herramientas es la capacidad de aprender de los datos y no solo de ejecutar unas funciones para las que están programadas. En el centro de estos sistemas se hallan los algoritmos de aprendizaje, que estudian los datos que reciben, encuentran patrones dentro de ellos y definen cuál es el comportamiento a seguir dependiendo de la situación. Esto es el *Aprendizaje Automático*, más conocido como *Machine Learning*.

Esta área hace referencia a las diferentes ramas de la programación sobre un ordenador para alcanzar una solución a un problema determinado de manera automática. Dentro de este aprendizaje podemos encontrarnos con tres principales vías dependiendo de la intervención de programador y de la estructura de los datos o del problema:

- **Aprendizaje supervisado.** Abarca los sistemas de enseñanza con situaciones iniciales conocidas y estrategias deseadas ya estudiadas.
- **Aprendizaje no supervisado.** El aprendizaje se realiza mediante simulaciones para encontrar los resultados deseados, sin ningún conocimiento previo de la estructura del problema.
- **Aprendizaje por refuerzo.** Se lleva a cabo una interacción con el entorno, sin necesidad de conocimientos previos o resultados conocidos, salvo el objetivo final del problema y una señal de refuerzo, positiva o negativa, imitando así el aprendizaje humano.

Este último tipo de aprendizaje es el que vamos a tratar en este trabajo.

1.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo tiene su origen en la psicología del aprendizaje animal, en la teoría del aprendizaje por prueba y error. Este punto de partida sirvió para realizar los primeros trabajos en inteligencia artificial, donde varios investigadores comenzaron a explorar el aprendizaje de prueba y error como un principio de ingeniería. Las primeras investigaciones computacionales de este aprendizaje fueron realizadas por Minsky y por Farley y Clark, ambas en 1954. En los años sesenta, los términos "refuerzo" y "aprendizaje de refuerzo" se utilizaron por primera vez en la literatura de ingeniería, pero no fue hasta principios de la década de 1980 cuando se comenzó a desarrollar propiamente.

Desde el aprendizaje por refuerzo o *reinforcement learning*, el sistema de aprendizaje viene motivado por la situación descrita anteriormente: el programador no conoce completamente las acciones que debería llevar a cabo el programa o agente, tan solo es capaz de decidir cuándo se ha completado la

tarea o no. Por lo que no se le puede advertir al agente la secuencia de acciones a realizar, sino más bien si una secuencia de acciones fue o no fue suficiente para completar la labor.

Ante esta situación, el aprendizaje del agente se basa en la interacción con el entorno, es decir, efectúa acciones y aprende de los resultados obtenidos. Este enfoque de aprendizaje está inspirado en el comportamiento humano, en el que actúa en busca de una recompensa.

El funcionamiento del aprendizaje por refuerzo se sintetiza en la figura 1.1. El procedimiento paso a paso es el siguiente:

1. El agente interacciona con el entorno y realiza una acción
2. Esta acción afecta al estado del agente en el entorno
3. La recompensa se limita a una señal indicando la conveniencia o no de las acciones llevadas a cabo
4. Con esta señal buscamos mejorar el comportamiento posterior

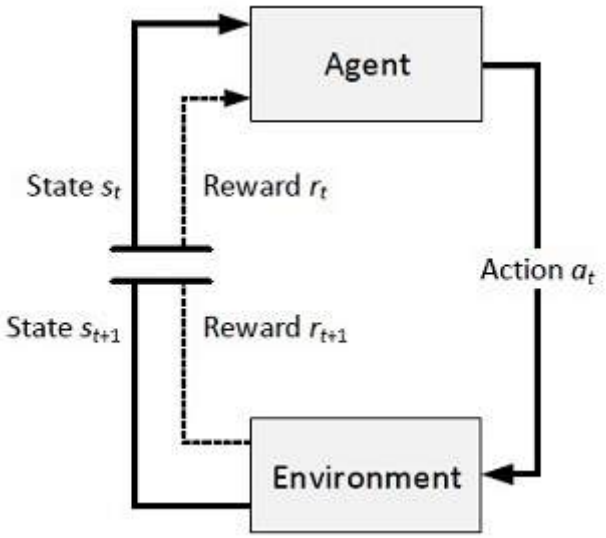

Figura 1.1: Esquema aprendizaje por refuerzo [8]

Como se ve en la figura anterior, en cada paso t el agente recibe una observación del ambiente s_t, ejecuta una acción a_t y recibe una recompensa r_t. El entorno que estaba en el estado s_t cambia por la acción a_t al estado s_{t+1} y emite la recompensa r_{t+1}.

En el aprendizaje por refuerzo se trata de encontrar un comportamiento óptimo para la resolución del problema aprendiendo de las acciones pasadas y maximizando la recompensa acumulada en el tiempo. Sin embargo, tal y como lo hemos presentado anteriormente, las recompensas no son instantáneas y queremos maximizar la acumulada, por lo tanto el agente debe razonar y elegir las acciones con las que a largo plazo obtenga mejor resultado.

Con el objetivo de establecer de manera correcta esta situación, el concepto de *procesos de decisión de Markov* va a ser esencial. La modelización del aprendizaje se desarrollará de acuerdo con estos procesos, ya que proporcionan un marco de trabajo sobre el que los métodos de *reinforcement learning* pueden ser contruidos.

Capítulo 2

Procesos de Decisión de Markov

En este capítulo presentaremos el espacio en el que se desarrolla el aprendizaje por refuerzo. Por lo visto en el capítulo anterior, este aprendizaje se basa en un proceso de interacción entre el agente y el entorno. Este proceso se suele enmarcar dentro de los llamados *Procesos de Decisión de Markov (MDP)*.

2.1. Definiciones principales

Los procesos de decisión de Markov se definen a partir de una tupla (S, A, T, R), donde:

- S es el conjunto no vacío de **estados**. El estado s_t representa el ambiente en el paso t.
- A es el conjunto no vacío de **acciones** posibles. En el paso t, el agente ejecuta la acción a_t.
- T es la distribución de probabilidad del conjunto:

  $$T : S\text{x}A\text{x}S \to \mathscr{P}(S)$$

  donde $T(s,a,s')$ es la probabilidad de que se realice una **transición** del estado s al s', ejecutando la acción a. En términos de probabilidades condicionadas, se puede reescribir de la siguiente manera: $T(s,a,s') = P(s'|(s,a))$
- R representa el **refuerzo** esperado dada una acción desde un estado concreto. Así pues R es la función:

  $$R : S\text{x}A\text{x}S \to \mathbb{R}$$

  de manera que $R(s,a,s') = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$, donde r_{t+1} es una variable aleatoria que corresponde a la recompensa recibida en la etapa $t+1$. En cada etapa el agente recibe un refuerzo dependiendo de la decisión tomada. El objetivo final es maximizar el refuerzo acumulado por el agente a lo largo de la vida del sistema.

Normalmente, en el ámbito del aprendizaje por refuerzo, se considera que el proceso de decisión de Markov en el que se enmarca el problema es finito, esto quiere decir que el conjunto de estados y de acciones es un conjunto finito. Consideraremos esta situación a lo largo del trabajo, y nos referiremos con el término *MDP* a un proceso de decisión de Markov finito.

Observar que la distribución de probabilidad definida con la función T solo depende del estado actual, del siguiente y de la acción a ejecutar. Es decir, la probabilidad de alcanzar el próximo estado s_i, depende tan solo del estado actual s_j y no de los estados que le preceden. Esta propiedad se llama *Propiedad de Markov*. Si se desea ampliar la información sobre procesos de Markov, se recomienda consultar el libro [12].

Ejemplo sencillo: LABERINTO 2 × 2

Uno de los ejemplos más sencillos para presentar los elementos descritos hasta ahora es un laberinto 2 × 2 como el siguiente:

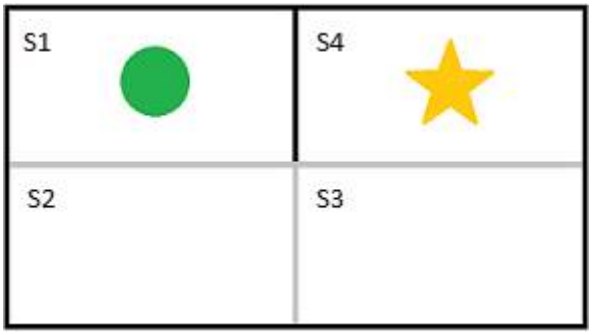

Figura 2.1: Esquema aprendizaje por refuerzo

En este ejemplo, las 4 casillas del laberinto forman el conjunto de estados posibles: $S = \{s_1, s_2, s_3, s_4\}$. *Las acciones posibles son subir, bajar, izquierda y derecha. Respectivamente llamamos el conjunto de acciones* $A = \{\text{“}up\text{”}, \text{“}down\text{”}, \text{“}left\text{”}, \text{“}right\text{”}\}$. *La distribución de probabilidad en este ejemplo es muy sencilla. Por ejemplo, si nos colocamos en las primera casilla* s_1 *y ejecutamos la acción subir, nos mantendremos en la misma casilla y no llegaremos nunca al estado* s_2, *por tanto* $T(s_1, \text{“}up\text{”}, s_2) = 0$. *De esta manera construimos cuatro matrices de probabilidades 4 × 4, una por cada estado inicial* s_1. *Así formamos el cuadro 2.1:*

Distribución	s_1	s_2	s_3	s_4
$T(s_1, \text{“}up\text{”}, s_i)$	1	0	0	0
$T(s_1, \text{“}down\text{”}, s_i)$	0	1	0	0
$T(s_1, \text{“}left\text{”}, s_i)$	1	0	0	0
$T(s_1, \text{“}right\text{”}, s_i)$	1	0	0	0
$T(s_2, \text{“}up\text{”}, s_i)$	1	0	0	0
$T(s_2, \text{“}down\text{”}, s_i)$	0	1	0	0
$T(s_2, \text{“}left\text{”}, s_i)$	0	1	0	0
$T(s_2, \text{“}right\text{”}, s_i)$	0	0	1	0

Distribución	s_1	s_2	s_3	s_4
$T(s_3, \text{“}up\text{”}, s_i)$	0	0	0	1
$T(s_3, \text{“}down\text{”}, s_i)$	0	0	1	0
$T(s_3, \text{“}left\text{”}, s_i)$	0	1	0	0
$T(s_3, \text{“}right\text{”}, s_i)$	0	0	1	0
$T(s_4, \text{“}up\text{”}, s_i)$	0	0	0	1
$T(s_4, \text{“}down\text{”}, s_i)$	0	0	1	0
$T(s_4, \text{“}left\text{”}, s_i)$	0	0	0	1
$T(s_4, \text{“}right\text{”}, s_i)$	0	0	0	1

Cuadro 2.1: Matrices de probabilidades

Observar que la probabilidad de llegar a un estado depende tan solo del estado actual y no de la secuencia de estados recorridos hasta la etapa actual. Es decir, que este ejemplo cumple la Propiedad de Markov.

El refuerzo lo establecemos de la siguiente manera: negativo (valor -1) si el estado al que llega es s_1, s_2 *o* s_3, *y positivo (+10) si es* s_4. *De esta manera, el programa intentará encontrar el itinerario más directo para llegar a la meta. Como los valores posibles de las probabilidades de transición son tan solo* 0 *y* 1, *el cálculo del refuerzo esperado* $R(s,a,s')$ *es muy sencillo:*

$$R(s_j, a_k, s_i) = \begin{cases} -1 & \text{si } i \neq 4 \\ 10 & \text{si } i = 4 \end{cases}$$

siendo $a_k \in A$ *cualquier acción.*

2.2. Desarrollo de los MDP

Así pues, en cada etapa el agente recibe un refuerzo dependiendo de la decisión tomada. El objetivo final es maximizar el refuerzo acumulado por el agente a lo largo de la vida del sistema. La recompensa acumulada la podemos presentar como sigue:

$$\sum_{k=0}^{\infty} \gamma^k R_k \tag{2.1}$$

donde $R_k = R(s_k, a_i, s_{k+1})$ es el refuerzo recibido en la etapa k. El parámetro γ es un **factor de descuento** ($0 \leq \gamma \leq 1$) en el que, salvo en el caso $\gamma = 1$, disminuye el refuerzo recibido por cada etapa que pasa. Este parámetro controla la importancia de los refuerzos a largo plazo: cuanto menor es el valor, menos peso se le otorga a las recompensas de las últimas etapas y más a las primeras, y viceversa.

En este sentido, la recompensa esperada se calcula para un número infinito de pasos pero favoreciendo la recompensa a corto plazo. Es por eso que este enfoque de la recompensa esperada se denomina *modelo de recompensa descontada en horizonte infinito*.

Existen otros criterios para formular la recompensa acumulada como el *modelo de recompensa en horizonte finito* y el *modelo de recompensa media* entre otros. En el primero se considera que solo se dispone de un número finito de pasos en los que se reciben recompensas, y por tanto la recompensa acumulada solo se calcula en esos pasos.

$$E\{\sum_{k=0}^{M} r_{t+k+1} | s_t = s\}$$

El modelo de recompensa media trata de maximizar la recompensa media a largo plazo.

$$\lim_{M \to \infty} E\{\frac{1}{M} \sum_{k=0}^{M} r_{t+k+1} | s_t = s\}$$

En el aprendizaje por refuerzo el modelo que usualmente se utiliza es el modelo de recompensa descontada en horizonte infinito (2.1). En este trabajo consideramos también esta alternativa.

La solución del problema vendrá dada como la asociación de una acción a cada estado de manera que se alcance el objetivo según el modelo de recompensa descontada en horizonte infinito. Esta asociación que define el comportamiento del agente se denomina política. Denotamos por tanto, la función **política**:

$$\pi : S \to A$$

La política de asociación de cada estado a una acción, solo depende del estado actual y no de los estados y las acciones pasadas (es decir, que cumple la *Propiedad de Markov*).

Presentamos a continuación dos conceptos importantes en la búsqueda de la optimalidad de la política:

- **Función de valor-estado.** Representa el refuerzo esperado desde el estado s y siguiendo la política π, $V_\pi : S \to \mathbb{R}$.

  $$V_\pi(s) = E[\sum_{k=0}^{\infty} \gamma^k r_{k+1} | s_0 = s, \pi]$$

  Esta función cuantifica la bondad de esta política para el caso particular de $s_0 = s$.

- **Función de valor-acción.** Representa el refuerzo esperado empezando desde el estado s, tomando la acción a y siguiendo con la política π, $Q_\pi : S \times A \to \mathbb{R}$.

  $$Q_\pi(s,a) = E[\sum_{k=0}^{\infty} \gamma^k r_{k+1} | s_0 = s, a_0 = a, \pi]$$

  Esta función explica la bondad de esta política para el caso particular de la pareja estado-acción ($s_0 = s$, $a_0 = a$).

De la función valor-estado podemos derivar una relación recursiva siguiente:

$$\begin{aligned}
V_\pi(s) &= E\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s,\pi\} \\
&= E\{r_{t+1}+\gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s,\pi\} \\
&= \sum_a p(s,a)[E\{r_{t+1}+\gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s,a_t=a,\pi\}] \\
&= \sum_a p(s,a)\sum_{s'}T(s,a,s')[E\{r_{r+1}|s_t=s,a_t=a,s_{t+1}=s'\}+\gamma E\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s',\pi\}] \\
&= \sum_a p(s,a)\sum_{s'}T(s,a,s')[R(s,a,s')+\gamma E\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s',\pi\}] \\
&= \sum_a p(s,a)\sum_{s'}T(s,a,s')[R(s,a,s')+\gamma V_\pi(s')]
\end{aligned}$$

En este desarrollo, la expresión $p(s,a)$ representa la probabilidad de tomar la acción a desde el estado s. El resultado es una fórmula recursiva denominada *ecuación de Bellman*:

$$V_\pi(s)=\sum_a p(s,a)\sum_{s'}T(s,a,s')[R(s,a,s')+\gamma V_\pi(s')] \tag{2.2}$$

Aplicando el mismo razonamiento del desarrollo anterior para la función $Q_\pi(s,a)$, obtenemos la siguiente expresión:

$$Q_\pi(s,a)=\sum_{s'}T(s,a,s')[R(s,a,s')+\gamma V_\pi(s')] \tag{2.3}$$

Y por lo tanto, tenemos la siguiente relación entre las funciones valor-estado y valor-acción:

$$V_\pi(s)=\sum_a p(s,a)Q_\pi(s,a) \tag{2.4}$$

Observar que la expresión descrita como la ecuación de Bellman (2.2), es un sistema lineal de ecuaciones. Reformulando la expresión podemos denominar $\mathbf{V}$ un vector columna formado por los valores de V_π en cada estado s, $\mathbf{R}$ un vector columna de longitud el número de estado en el que cada componente es

$$\mathbf{R}_i = \sum_a p(s_i,a)\sum_{s'}T(s_i,a,s')R(s_i,a,s')$$

y $\mathbf{P}$ es una matriz $|S|\times|S|$ dada por

$$\mathbf{P}_{i,j} = \sum_a p(s_i,a)T(s_i,a,s_j)$$

Finalmente obtenemos el sistema lineal:

$$\mathbf{V}=\mathbf{R}+\gamma\mathbf{P}\mathbf{V} \tag{2.5}$$

cuya solución viene dada por

$$\mathbf{V}=(\mathbf{I}-\gamma\mathbf{P})^{-1}\mathbf{R}$$

Teorema 1. *El sistema* $\mathbf{V}=(\mathbf{I}-\gamma\mathbf{P})^{-1}\mathbf{R}$ *tiene solución única, con* $\gamma<1$.

Demostración. Por construcción, la matriz $\mathbf{P}$ es estocástica ya que sus filas representan probabilidades de transición del estado correspondiente a la fila al de la columna. Por tanto, se tiene que

$$\| \mathbf{P} \|_\infty = \max_{s_i} \sum_{s_j} |\mathbf{P}_{i,j}| = 1$$

Esto implica que $\| \gamma\mathbf{P} \|_\infty = \gamma < 1$. Los valores propios de la matriz $\mathbf{P}$ son menores o iguales que 1 y por tanto la matriz $\mathbf{I} - \gamma\mathbf{P}$ es invertible. Así pues, el sistema tiene solución única. □

2.3. Política óptima

Como hemos descrito anteriormente, el objetivo del desarrollo de los MDP es encontrar una política que defina el comportamiento de agente y maximice la recompensa a largo plazo. Esta política, π^*, la denominaremos *política óptima*. Por las definiciones de las funciones valor-estado y valor-acción, se deduce que esta política óptima obtendrá los valores más altos posibles. En el caso de la función V esto significa que $V_{\pi^*}(s) \geq V_\pi(s)$, para todo estado s. Llamamos a este valor máximo de la función como:

$$V^*(s) = \max_\pi V_\pi(s) \qquad \forall s \in S$$

Análogamente para la función valor-acción se define:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \qquad \forall s \in S, \forall a \in A$$

Podemos encontrar una relación entre ambas funciones óptimas desde la relación anterior (2.4). Observar que

$$V_\pi(s) = \sum_a p(s,a) Q_\pi(s,a) \leq \max_a Q_\pi(s,a)$$

En particular, para el caso óptimo tenemos que $V^*(s) \leq \max_a Q^*(s,a)$. Si para ciertos $s \in S$, se diera la desigualdad, la elección de la acción a daría un valor de V^* mejor, lo que es imposible ya que es la óptima. Por lo tanto, tenemos que:

$$V^*(s) = \max_a Q^*(s,a) \qquad \forall s \in S$$

Para el caso de las ecuaciones de Bellman (2.2) y (2.3) en la política óptima, se obtienen las conocidas como *ecuaciones óptimas de Bellman*:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{2.6}$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

Esta última ecuación permite encontrar la política óptima si se conocen los valores óptimos de la función Q. La asociación adecuada de la acción a cada estado será la acción que haga máximo el valor de esta función, por lo tanto:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s,a)$$

Así, la política óptima viene dada por las funciones óptimas $V^*(s)$ y $Q^*(s,a)$. Por tanto, podemos resolver el problema conociendo los valores de las funciones óptimas.

Para el cálculo de la función V^*, podemos optar por reescribir la expresión (2.6) como sistema de ecuaciones de manera análoga a la del apartado anterior (2.5), salvo que ahora no tenemos un sistema lineal ya que aparece la función no lineal máx. Atendiendo a esta salvedad, construímos las matrices

siendo las componentes del vector $\mathbf{R}^a$ de la manera $\mathbf{R}^a_i = \sum_{s'} T(s_i, a, s')R(s_i, a, s')$ y las de $\mathbf{P}^a$ tal que $\mathbf{P}^a_{i,j} = T(s_i, a, s_j)$

$$\mathbf{V}^* = \max_a \mathbf{R}^a + \gamma \mathbf{P}^a \mathbf{V}^*$$

Con el objetivo de resolver este sistema, el cálculo directo puede ser costoso al no tratarse de un sistema lineal. Por ello, se suele preferir resolverse por métodos iterativos:

$$\mathbf{V}^* \leftarrow \max_a \mathbf{R}^a + \gamma \mathbf{P}^a \mathbf{V}^* \tag{2.7}$$

Una vez planteado este método iterativo para calcular el valor de V^*, se plantean cuestiones sobre la convergencia del método y su comportamiento. Para responder a estos temas definimos lo siguiente:

Definición 1. Llamamos *operador de Bellman* al operador dado por

$$H^*(\mathbf{V}) = \max_a \mathbf{R}^a + \gamma \mathbf{P}^a \mathbf{V}$$

Existen diferentes definiciones del operador de Bellman. En esta ocasión, la definición que más se ajusta a nuestro contexto es la presentada anteriormente. Para llegar a probar la convergencia del método iterativo necesitamos la siguiente propiedad:

Proposición 1. *El operador de Bellman es una contracción, es decir,*

$$\| H^*(\widehat{\boldsymbol{V}}) - H^*(\boldsymbol{V}) \|_\infty \leq \gamma \| \widehat{\boldsymbol{V}} - \boldsymbol{V} \|_\infty$$

Demostración. Contenida en el Anexo 1 □

Teorema 2. *Para cualquier valor inicial $\boldsymbol{V}_0$, la iteración dada por*

$$H^*(\boldsymbol{V}_n) = \boldsymbol{V}_{n+1}$$

converge a $\boldsymbol{V}^*$.

Demostración. Contenida en el Anexo 1. □

Así pues, hemos conseguido un método iterativo para el cálculo del valor óptimo de V.

Capítulo 3

Algoritmos de aprendizaje. Q-learning

Los algoritmos necesarios para el aprendizaje tienen el objetivo de encontrar la política óptima y maximizar los valores de las funciones anteriores. Este aprendizaje puede enfocarse de dos maneras diferentes dependiendo de la información que se tenga sobre el problema. Concretamente, la incertidumbre en el conjunto de los datos la encontramos en la distribución de probabilidades y/o en la de las recompensas, ya que los estados y las acciones han de ser conocidas para entender el planteamiento del problema. En el ejemplo presentado anteriormente, sabemos perfectamente la probabilidad de llegar a un cierto estado comenzando en otro y ejecutando cierta acción, pero no siempre es posible tener a nuestra disposición toda la información. De hecho, en las mayores aplicaciones de este tipo de aprendizaje no conocemos el escenario completo, y es aquí donde reside el interés del aprendizaje por refuerzo. Presentamos a continuación las dos posibilidades.

3.1. Aprendizaje basado en modelo

El modelo del ambiente es un MDP con la distribución de probalidades y recompensas conocidas. El aprendizaje en este caso consiste en llegar a entender el MDP y resolverlo, mediante los algoritmos que nos proporciona la programación dinámica. Mediante el estudio de este aprendizaje, se han desarrollado varios métodos de resolución, pero los dos clásicos son el de *iteración de valores* y el de *iteración de políticas*.

3.1.1. Iteración de valores

Como hemos citado anteriormente, este método tiene como objetivo encontrar una política óptima y la manera de encontrarla es lo que difiere entre los métodos. En este caso, la búsqueda se realiza mediante el cálculo de el valor óptimo para la función valor-estado.

Para obtener el resultado, debemos ir actualizando el valor de la función $V(s)$ utilizando la ecuación óptima de Bellman (2.6) descrita en el capítulo anterior:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

El método de acualización construye una aproximación V_{k+1} de V en la etapa $k+1$ usando el valor V_k de V de la etapa anterior. De esta manera, por cada iteración vamos mejorando el valor de la función. En esencia, se trata del método iterativo (2.7) descrito en el capítulo anterior.

El criterio para finalizar el algoritmo es el *criterio residual de Bellman*, en el que se fija el final de la iteración cuando la máxima diferencia en cada estado entre los valores de la función de un paso y el siguiente es menor que ε:

$$\max_{s \in S} |V_{k+1}(s) - V_k(s)| < \varepsilon$$

En términos de las matrices presentadas anteriormente, este criterio de parada se puede reescribir como:

$$\| \mathbf{V}_{k+1} - \mathbf{V}_k \|_\infty < \varepsilon$$

Si se da esta situación, el algoritmo parará y nuesto valor óptimo de V será V_{k+1}. Procede ahora buscar el error cometido por este criterio, es decir, encontrar a qué distancia se halla el valor óptimo V^* de nuesto valor V_{k+1}. Para tratar esta cuestión recurriremos al operador de Bellman.

Teorema 3. *Si* $\| H^{*(k+1)}(V) - H^{*(k)}(V) \|_\infty < \varepsilon$*, entonces*

$$\| V^* - H^{*(k+1)}(V) \|_\infty < \frac{\varepsilon}{1-\gamma}$$

Demostración. En esta notación, $\| H^{*(k+1)}(\mathbf{V}) - H^{*(k)}(\mathbf{V}) \|_\infty = \| \mathbf{V}_{k+1} - \mathbf{V}_k \|_\infty < \varepsilon$. Podemos escribir el vector óptimo de V como la infinitésima iteración del operador de Bellman de manera que $\mathbf{V}^* = H^{*(\infty)}(\mathbf{V})$. Así pues tenemos que:

$$\begin{aligned}
\| \mathbf{V}^* - H^{*(k+1)}(\mathbf{V}) \|_\infty &= \| H^{*(\infty)}(\mathbf{V}) - H^{*(k+1)}(\mathbf{V}) \|_\infty = \\
&= \| \sum_{t=1}^{\infty} H^{*(t+k+1)}(\mathbf{V}) - H^{*(t+k)}(\mathbf{V}) \|_\infty \leq \\
&\leq \sum_{t=1}^{\infty} \| H^{*(t+k+1)}(\mathbf{V}) - H^{*(t+k)}(\mathbf{V}) \|_\infty = \\
&= \sum_{t=1}^{\infty} \gamma^t \varepsilon < \frac{\varepsilon}{1-\gamma}
\end{aligned}$$

□

El algoritmo completo de la iteración de valores se describe en el algoritmo 1.

Algorithm 1 Iteración de valores

```
Inicializamos V(s) arbitrario
repeat
    δ ← 0
    for all s ∈ S do
        v ← V(s)
        V(s) ← máx_a Σ_s' T(s,a,s')[R(s,a,s') + γV(s')]
        δ ← máx(δ, |v − V(s)|)
    end for
until δ < ε
```

La iteración de valores asegura la convergencia sobre una política óptima en un número finito de pasos, incluso antes de alcanzar el valor óptimo para la función V. La convergencia de este método está condicionada a la visita de todos los estados varias veces.

3.1.2. Iteración de políticas

Este segundo método modifica directamente la política utilizada en lugar de mejorar la función valor-estado. Iniciando desde una política π_0, se trata de evaluar esta política y mejorarla si es posible. Estos son los dos fases que tiene este método.

Más concretamente, el primer paso consiste en evaluar la política calculando el valor V^{π_0} de manera similar a la iteración anterior, salvo que en este caso, solo evaluamos con la ecuación de Bellman (2.2), no con la óptima (2.6).

$$V_{k+1}^{\pi}(s) \leftarrow \sum_a p(s,a) \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k^{\pi}(s')]$$

El critero de parada en este caso es el final de la resolución del sistema de ecuaciones lineales (observar que antes no se podía por tener el operador máx).

La fase de mejora de la política se caracteriza por la comparación entre la política actual y la calculada como la mejor para esta función valor-estado. Se puede probar que esta política siempre mejorará la actual, a menos que la actual sea ya la óptima.

Teorema 4. *Sea la sucesión* $(\boldsymbol{V}^n)$ *los valores de* $\boldsymbol{V}$ *cada iteración n del algoritmo de iteración de políticas. Entonces se tiene que*

$$\boldsymbol{V}^n \leq \boldsymbol{V}^{n+1} \leq \boldsymbol{V}^*$$

Demostración. Contenida en el Anexo 1. □

Observación. Dos valores consecutivos de $\mathbf{V}$ son iguales solo en la última iteración. Además, el número total de posibles políticas es $|A|^{|S|}$, luego este es el número máximo de iteraciones.

El desarrollo del algoritmo está presentado en el algoritmo 2.

Algorithm 2 Iteración de políticas

Inicializamos $V(s)$ y $\pi(s)$ arbitrarios
repeat
  EVALUACIÓN DE LA POLÍTICA
  Resolver el sistema
  $V(s) = \sum_a p(s,a) \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
  MEJORA DE LA POLÍTICA
  $politicaoptima? \leftarrow VERDADERO$
  for all $s \in S$ **do**
    $b \leftarrow \pi(s)$
    $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
    if $b \neq \pi(s)$ **then**
      $politicaoptima? \leftarrow FALSO$
    end if
  end for
until $politicaoptima? = VERDADERO$

El método de iteración de políticas converge en un número menor de iteraciones respecto el de iteración de valores, aunque en cada iteración de este método tiene un mayor coste computacional por la resolución de un sistema lineal. En definitiva, no existe una preferencia clara de un método iterativo sobre otro.

3.2. Aprendizaje sin modelo

En este caso, el modelo del ambiente no es conocido completamente (por ejemplo, se desconocen las probabilidades de transición o los refuerzos). Ahora se trata de encontrar la política óptima, sin llegar a especificar completamente el modelo, mediante simulaciones del problema y su interacción con el ambiente, aprendiendo de la experiencia. Este modelo tiene varias ventajas con respecto al anterior, ya que muchas veces las probabilidades de transición o las recompensas son desconocidas. Los algoritmos utilizados para este aprendizaje mediante modelos libres se conocen como *métodos de diferencia temporal*, aunque el más importante es el algoritmo *Q-learning* o *Q-algoritmo*.

3.2.1. Métodos de diferencia temporal: TD(0)

Los métodos de diferencia temporal son algoritmos de aprendizaje cuya característica en común es que utilizan estimaciones de funciones valor-estado de otros estados para actualizar el estado acutal. El método más sencillo *TD(0)* fue propuesto por *Sutton (1998)* [11], y se rige por:

$$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$$

A través de la evaluación de la política π sobre el estado s, observamos el estado resultante s' y la recompensa r. La expresión $r + \gamma V(s')$ funciona como estimador del valor de $V(s)$. Observar que, considerando la fórmula (2.2) de esta función, y tomando $p(s,a) = 1$ para la acción elegida por π y $T(s,a,s') = 1$ para el estado de llegada observado s', esta fórmula se reduce a la expresión $r + \gamma V(s')$. Es decir, que el término $r + \gamma V(s')$ es un estimador de $V(s)$ si tuvieramos la certeza de que se ejecuta esta acción a y el nuevo estado es s'. Como la actualización de $V(s)$ se realiza con cada uno de los siguientes estados, este criterio corresponderá con una buena estimación.

El parámetro $\alpha \in (0,1]$ es un factor de aprendizaje en el que se nivela si se le da más o menos valor al aprendizaje que se tenía o al que se adquiere nuevo. Este factor α se conoce como *learning rate*. El algoritmo completo está recogido en el algoritmo 3.

Algorithm 3 TD(0)

1: Inicializamos $V(s)$ arbitrario
2: Elegir la política π a evaluar
3: Inicializar s
4: **loop**
5: $a \leftarrow$ acción probable por $\pi(s)$
6: Tomando a, observar la recompensa r y el siguiente estado s'
7: $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$
8: $s \leftarrow s'$
9: **end loop**

Los métodos de diferencia temporal se dividen en dos clases, distinguiéndose por una caracterísitica importante:

- *Métodos on-policy:* la política de control del problema MDP es la misma que es evaluada y mejorada en el propio método.
- *Métodos off-policy:* la política de control puede no tener relación con la política empleada en el algoritmo.

A continuación presentaremos los dos principales algoritmos para estos dos métodos de diferencia temporal: *Q-learning* y *Sarsa*.

3.2.2. Q-learning: método *off-policy*

El método *Q-learning* fue propuesto por *Watkins (1989)* [14]. Es el procedimiento más importante en cuanto a aprendizaje por refuerzo en modelos libres.

Este algoritmo utiliza el conocimiento actual a la vez que explora el entorno. Concretamente, en cada paso mira para el estado siguiente el máximo refuerzo posible para todas las acciones posibles en ese estado, y utiliza esta información para actualizar el conocimiento sobre la pareja acción-estado para el correspondiente acción del estado actual. La actualización del conocimiento se realiza de la siguiente manera:

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

donde γ es un factor de descuento, $max_{a'}Q(s',a')$ es el valor óptimo esperado. Como anteriormente la expresión $r' + \gamma.max_{a'}Q(s',a')$ es una estimación de $Q(s,a)$.

Como se puede ver, en este caso se busca la optimalidad de la función $Q(s,a)$ de manera que ésta pueda determinar directamente la política óptima. En el algoritmo 4 se recoge el desarrollo de este método.

Algorithm 4 Q-learning

1: Inicializamos $Q(s,a)$ arbitrario
2: Inicializar s
3: **loop**
4: $a \leftarrow$ acción probable por $\pi(s)$ derivada de $Q(s,a)$
5: Tomando a, observar la recompensa r y el siguiente estado s'
6: $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a))$
7: $s \leftarrow s'$
8: **end loop**
9: $\forall s \in S$, $\pi(s) \leftarrow argmax_a Q(s,a)$

Para que se dé la convergencia del método, toda pareja estado-acción ha de ejecutarse infinitas veces. El algoritmo Q-learning admite que la tasa de aprendizaje α sea no constante y que varíe en función de la etapa t, α_t. El siguiente resultado nos proporciona una condición suficiente para garantizar la convergencia en el caso general.

Teorema 5. *El algoritmo Q-learning converge con probabilidad 1 al valor óptimo Q^* si el parámetro α cumple*

$$\sum_t \alpha_t = \infty \qquad \sum_t \alpha_t^2 < \infty$$

Demostración. Contenida en el Anexo 1. □

Si se desea ampliar la información sobre la convergencia del algoritmo Q-learning se recomienda el breve artículo [3] y sus referencias.

3.2.3. Sarsa: método *on-policy*

El método *Sarsa* fue introducido por primera vez por *Rummery y Niranjan (1994)*, sección 6.4 [11]. En este algoritmo, la política que se utiliza para elegir las acciones es la misma que se evalua y mejora. Por lo tanto, el núcleo del algoritmo queda como:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

Notar que se diferencia del *Q-learning* por la ausencia el operador *max*, lo que indica que se está utilizando la misma política para la estimación de $Q(s,a)$. En cada iteración se necesita la información de s,a,r,s',a', de ahí el nombre del método. El algoritmo completo está presentado en el algoritmo 5.

Algorithm 5 Sarsa

1: Inicializamos $Q(s,a)$ arbitrario
2: Inicializar s
3: $a \leftarrow$ acción probable por $\pi(s)$ derivada de $Q(s,a)$
4: **loop**
5: Tomando a, observar la recompensa r y el siguiente estado s'
6: $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$
7: $s \leftarrow s'$, $a \leftarrow a'$
8: **end loop**

3.3. Exploración – Explotación

Cuando nos encontramos ante un algoritmo de aprendizaje por refuerzo, se plantea la cuestión de cómo debe aprender el método. Este asunto se debate entre la alternativa de usar la información ya adquirida o intentar conseguir nuevos datos. En primera instancia, tenemos una política inicial que dicta las acciones a ejecutar en cada estado con cierta probabilidad. Si el algoritmo se limita a la ejecución y mejora de esa política, lo que se llama *explotación*, su aprendizaje es limitado. Ha de implementarse un medio de manera que ciertos comportamientos inaccesibles y a priori inadecuados desde esa política, puedan ser posibles. Este es el concepto de *exploración*. En el aprendizaje por refuerzo, los algoritmos deben actuar mientras aprenden, deben ser una combinación de ambos conceptos.

En este sentido, la capacidad de controlar esta combinación en los algoritmos descritos anteriormente reside en la tasa de aprendizaje α. Con este parámetro, podemos nivelar la veracidad que concedemos a la estimación de la función V ó Q, dependiendo de la iteración, en la actualización de los valores de las funciones.

Además de una mejora en el aprendizaje, la exploración es un concepto necesario en los métodos anteriores. La condición necesaria para la convergencia de cualquier método es que todos los estados sean visitados infinitas veces, es decir, que ningún estado tenga una probabilidad cero de ser visitado. Cuando la situación es esta, la exploración se hace imprescindible.

Asimismo, podemos modificar las probabilidades de transición para no encontrarnos con la situación anterior. Esta idea ha generado diferentes métodos para dar lugar a la exploración desde las propias probabilidades de transición. La combinación de explotación-exploración desde esta perspectiva la podemos abordar de varias maneras, los dos métodos más importantes son:

- **ε-greedy:** Esta es la opción más comunmente utilizada. Tomamos un parámetro $\varepsilon \in (0,1)$ para modificar las probabilidades de transición en las acciones, de manera que ningún par estado-acción resulta con probabilidad cero de ser visitado. Esta política se describe como sigue:

$$p(s,a) = \begin{cases} \frac{\varepsilon}{|A|} & \text{si } a \text{ es una acción de exploración} \\ 1 - \varepsilon + \frac{\varepsilon}{|A|} & \text{otro caso} \end{cases}$$

- **Softmax:** Algunas de las acciones de exploración, pueden ser muy adecuadas, sin embargo hemos dado la misma probabilidad a todas. Para solucionar este problema, determinamos la probabilidad de cada acción basándonos en la magnitud relativa de las estimaciones de la función Q, es decir:

$$p(s,a) = \frac{exp(Q(s,a)/\tau)}{\sum_{a'} exp(Q(s,a')/\tau)}$$

  El parámetro *temperatura*, τ, determina cómo de arbitrario es la selección de la acción (mayor valor del parámetro, mayor arbitrariedad). A su vez, la función $Q(s,a)$ define qué acciones con altos valores de la funcion Q, tendrán mayor probabilidad de ser seleccionadas.

  Este método está basado en una distribución de probabilidad procedente del estudio físico de partículas en un sistema de diferentes estados, *la distribución de Boltzmann*. Esta distribución proporciona la probabilidad de que un sistema esté en un estado en función de la energía de ese estado y de la temperatura del sistema.

En resumen, en los métodos y algoritmos más extendidos, el control sobre explotación - exploración reside en la tasa de aprendizaje α y en los parámetros ε y τ cuando nos encontremos en el método ε-greedy o softmax respectivamente. En el capítulo siguiente será con estos parámetros con los que estudiaremos diferentes resultados variando sus valores.

Capítulo 4

El aprendizaje por refuerzo en *R*

Para comenzar a trabajar con el programa *R* en el ámbito del aprendizaje por refuerzo, debemos examinar las herramientas disponibles para estudiar este tipo de problemas. Existen principalmente dos paquetes en *R* para abordar las cuestiones que hemos descrito en los capítulos anteriores: *MDPtoolbox* y *ReinforcementLearning* [9].

El primero se limita al tratamiento de problemas que se encuentran dentro de los Procesos de Markov. Las estrategias que sigue son las de programación dinámica, es decir, requiere toda la información del problema incluidas acciones y recompensas. Esta biblioteca alberga ciertas funciones que permiten al usuario proceder a la resolución del problema mediante el método de *iteración de valores* y mediante el de *iteración de políticas*.

En cambio, el paquete *ReinforcementLearning* funciona como el modelo libre del aprendizaje por refuerzo propiamente descrito, hace posible el aprendizaje del programa basándose en secuencias de simulaciones.

A continuación detallaremos ambas bibliotecas y las funciones que contienen.

4.1. Biblioteca *MDPtoolbox*

Para resolver cierto problema mediante este paquete, debemos tener disponible toda la información del problema. Hemos de definir todos los estados y las acciones posibles, así como todas las matrices con las probabilidades de transición entre estados y las recompensas (positivas y negativas). Es decir, este paquete está orientado a realizar un aprendizaje del sistema basado en modelo. El ejemplo propuesto en el capítulo 2, *Laberinto 2×2*, podríamos definirlo en el programa y trabajarlo desde esta herramienta. Examinaremos las diferentes herramientas a través del desarrollo de un ejemplo similar.

Ejemplo. GRID 3×4

En este ejemplo sencillo el objetivo es llegar a la casilla verde, sin caer en la roja. A la casilla negra no se puede llegar.

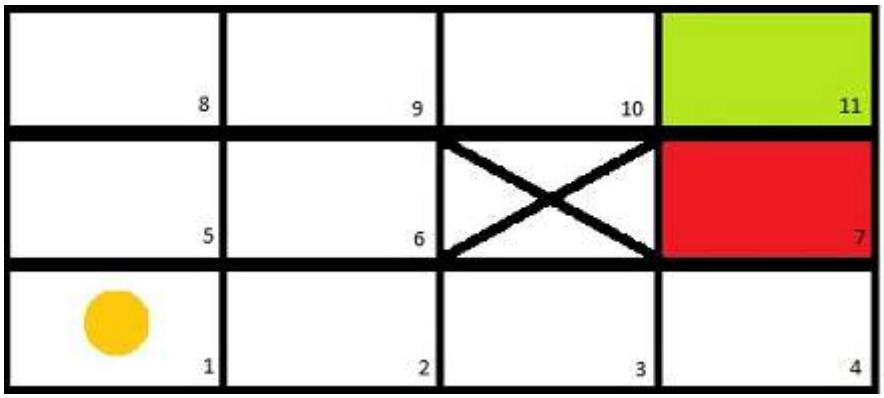

Figura 4.1: Grid 3×4

Para tratar este problema como un MDP, establecemos el marco del problema definido: los estados como las 11 posibles casillas y las acciones como *"up"."left","down","right"*. Para la distribución de las probabilidades de transición T debemos generar 4 matrices, una por acción. A diferencia del ejemplo *Laberinto 2×2*, vamos a permitir la posibilidad de "errores" en la ejecución de las acciones, es decir, si estamos en el estado 1 y ejecutamos la acción "*up*", con una probabilidad de 0,8 pasaremos al estado 5, con un 0,1 al 2 y con un 0,1 al 6. De esta manera damos pie a cierta exploración en el algoritmo del problema markoviano. Estas matrices tienen dimensión 11×11, ya que son transiciones de estado a estado. Para los refuerzos, como van asociados a una pareja estado-acción, definiremos una matriz 11×4 de manera que si el estado de llegada es el mismo que el de partida, la recompensa sea 0, en caso contrario, se sigue el criterio siguiente:

$$r(a_k,s_i)=\begin{cases}-1 & \text{si } i\neq 7, i\neq 11\\ -100 & \text{si } i=7\\ 100 & \text{si } i=11\end{cases}$$

siendo a_k cualquier acción. El desarrollo del planteamiendo se adjunta en el Anexo 2.

Una vez definido en el programa el espacio del problema, estamos en condiciones de utilizar los materiales que nos proporciona la biblioteca *MDPtoolbox*. En primer lugar, ejecutaremos la función `mdp_check`, que simplemente comprueba que el problema está bien definido como un MDP. En caso correcto, devuelve una respuesta vacía o describe el error en caso contrario.

Para la resolución del problema mediante el método de iteración de políticas existe la función `mdp_policy_iteration`, que requiere las matrices de transición P, la de recompensas R y el factor de descuento, y el programa nos devuelve lo siguiente:

```
>m <- mdp_policy_iteration(P=T, R=R0, discount=0.9)
>m

$V
 [1]  5.767969e+01  5.547313e+01  1.361721e+01  1.055427e+01  6.450564e+01  7.158948e+01
 [7]  0.000000e+00  7.160708e+01  8.271796e+01  9.587320e+01 -1.973730e-14

$policy
 [1] 1 1 4 2 1 1 1 4 4 4 1

$iter
[1] 2

$time
Time difference of 0.1184442 secs
```

Obtenemos los valores óptimos de la función V para cada estado, $V^*(s)$, luego tiene 11 componentes. El término `policy` se refiere a la política óptima, que consta de unos valores numéricos asociados a cada estado, luego son valores de 1 a 4 y tiene 11 componentes. En este caso se interpreta que en el estado 1 deberíamos tomar la acción número 1, "*up*", y así sucesivamente. Finalmente, el programa también nos devuelve el número de iteraciones y el tiempo utilizado para la ejecución de este método.

Consideramos ahora que las probabilidades de transición no permiten "errores", es decir, al ejecutar una acción la probabilidad de que se realice satisfactoriamente es 1. Obtenemos de nuevo la misma política aunque obtenemos nuevos valores de la función valor-estado:

```
>m0<- mdp_value_iteration(P=T0,R=R0, discount=0.9)
>m0$V[1]
[1] 62.171
```

Si calculásemos este mismo parámetro, es decir, la recompensa esperada en este estado $s1$ mediante la definición de modelo de recompensa descontada en horizonte infinito (2.1), obtenemos el siguiente valor (recordar que $\gamma = 0{,}9$):

$$\sum_{k=0}^{\infty} \gamma^k R_k = 1.(-1) + 0{,}9^1.(-1) + 0{,}9^2.(-1) + 0{,}9^3.(-1) + 0{,}9^4.(+100) = 62{,}171$$

Resulta el mismo valor obtenido mediante el procedimiento en *R*, como se debía esperar.

Podemos resolver el problema mediante el método de iteración de valores en lugar de por el de políticas. La función `mdp_value_iteration` nos permite proceder a la resolución de esta manera indicando los mismo datos que en el caso anterior: matrices de transición *P*, de recompensas *R* y factor de descuento. La interpretación de los resultados es la misma que anteriormente. Aunque en este caso aparece también el valor de ε y del factor de descuento por el que se ha procedido. El valor de γ lo habíamos introducido, en cambio el de ε si no se introduce, utiliza por defecto el valor 0,01.

Esta biblioteca también permite aplicar el operador de Bellman teniendo disponible las probabilidades de transición, las recompensas, el factor de descuento y un valor de *V* inicial. La función `mdp_bellman_operator` aplica una vez el operador descrito en el capítulo 2, sin el concepto de máximo.

En este paquete también podemos encontrar una función para tratar los problemas mediante el algoritmo *Q-learning*. Esta herramienta se encuentra un poco más alejada de las demás ya que se sale del marco de la programación dinámica. Requiere que se introduzcan los datos de las probabilidades de transición y las recompensas. El resultado es la matriz *Q*, una matriz $|S| \times |A|$ cuyos elementos son los valores de la función *Q* de la acción correspondiente a la fila y el estado de la columna. Esta orden también devuelve los valores de la función *V* final para cada estado y la política óptima obtenida.

El código completo del ejemplo *GRID* 3×4 para esta biblioteca se encuentra en el Anexo 2.

4.2. Biblioteca *ReinforcementLearning*

En este segundo paquete, debemos presentar el problema también como un problema de Markov, aunque no tenemos que conocer necesariamente las transiciones ni las recomensas. En todo caso, debemos definir el problema por los estados y las acciones posibles. Presentaremos los mecanismos de trabajo que nos permite esta librería mientras desarrollamos el estudio del ejemplo anterior.

Dado que no podemos definir el entorno de manera precisa como en el caso de la biblioteca anterior, surge la necesidad de simular intentos del problema para tener nociones de cómo es este entorno. Así pues, en primer lugar generamos una base de datos a través de varias simulaciones del juego. Esto lo conseguimos con la función `sampleExperience`, en la que debemos introducir el número de simulaciones que queremos hacer y el espacio del problema: los estados, las acciones, las recompensas y el funcionamiento del juego. En estas muestras, la manera de elegir la acción es aleatoria. Posteriormente veremos que podemos modificarlo.

```
data <- sampleExperience(N = 100, env = Pawn3x4Grid,
         states = allstates, actions = actions,
         design=tableau)
```

En nuestro caso, hemos generado el ambiente `Pawn3x4Grid` en el que se encuentra el funcionamiento del juego. El objetivo de las simulaciones es conseguir un conjunto de datos del que poder aprender a partir de él. La realización del propio aprendizaje sin modelo, es decir, a partir del conjunto de datos,

se ejecuta con la función `ReinforcementLearning`.

```
model0 <- ReinforcementLearning(data, s = "State", a = " Action ",
          r = " Reward ", s_new = "NextState", iter=1, control = control)
```

Esta función nos devuelve la matriz Q y la política óptima aprendida mediante *reinforcement learning*.

En esencia, estas son las herramientas principales de esta biblioteca. A continuación vamos a estudiar diferentes posibilidades y casos particulares en nuestro ejemplo.

Como hemos citado anteriormente, la eleccion de la acción es aleatoria por defecto. Pero también está implementada la opción de utilizar el método $\varepsilon - greedy$. Además, de manera propia hemos introducido la posibilidad de proceder mediante el método *softmax*. Estas opciones son seleccionadas en *R* mediante el siguiente argumento de la función `sampleExperience`:

`actionSelection= " epsilon-greedy "`

`actionSelection= " random "` (o simplemente no escribir nada)

Para el caso del *softmax*, hemos tenido que modificar la función `sampleExperience` para poder implementar esta posibilidad. A esta función modificada la hemos llamado `adquiereExperiencia`, sintácticamente se añade lo mismo que para el caso anterior y además ahora podemos introducir el atributo siguiente:

`actionSelection= "softmax"`

La gráfica que recoge la evolución de los valores de la función V para un estado concreto a lo largo de las simulaciones se denomina curva de aprendizaje o *learning curve*. Presentamos a continuación una gráfica comparativa de las curvas de aprendizaje en el estado $s = s1$ para los tres diferentes procedimientos anteriores aplicados a nuestro ejemplo.

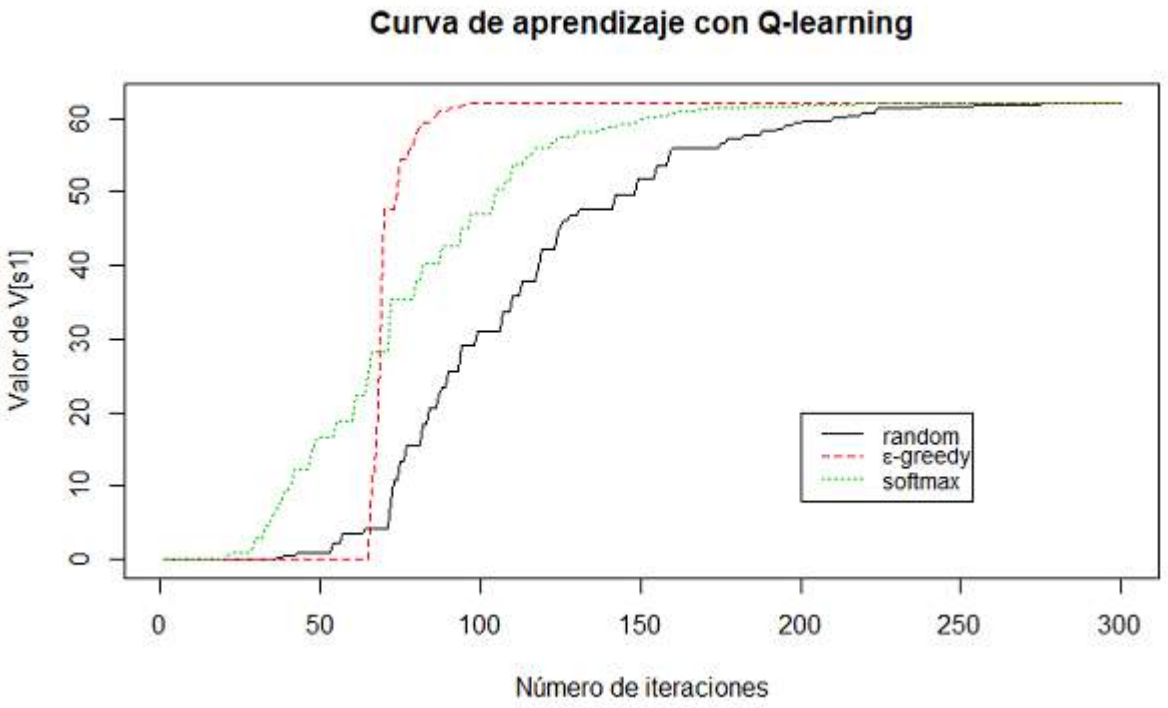

Figura 4.2: Comparación de la evolución de los valores de la función V para el estado 1.

Los valores de los parámetros de la gráfica 4.2 son $\varepsilon = 0{,}1$, $\tau = 100$ y la tasa de aprendizaje $\alpha = 0{,}1$. Aunque la máquina comienza a aprender antes con el método softmax, se puede observar un mejor aprendizaje para el caso ε-greedy, el valor de V crece con mayor rapidez. El valor óptimo de la función es de 62,171, como ya habíamos calculado mediante las funciones de la biblioteca *MDPtoolbox*. Como explicamos en la sección 3.3, si tomamos valores de τ más grandes, la gráfica verde se acerca a la negra, y se acerca a la roja si los tomamos más pequeños, incluso pudiendo llegar a mejorar el aprendizaje por ε-greedy.

Veamos ahora en la figura 4.3 los diferentes aprendizajes para valores distintos de la tasa de aprendizaje, solo para el caso del método $\varepsilon - greedy$.

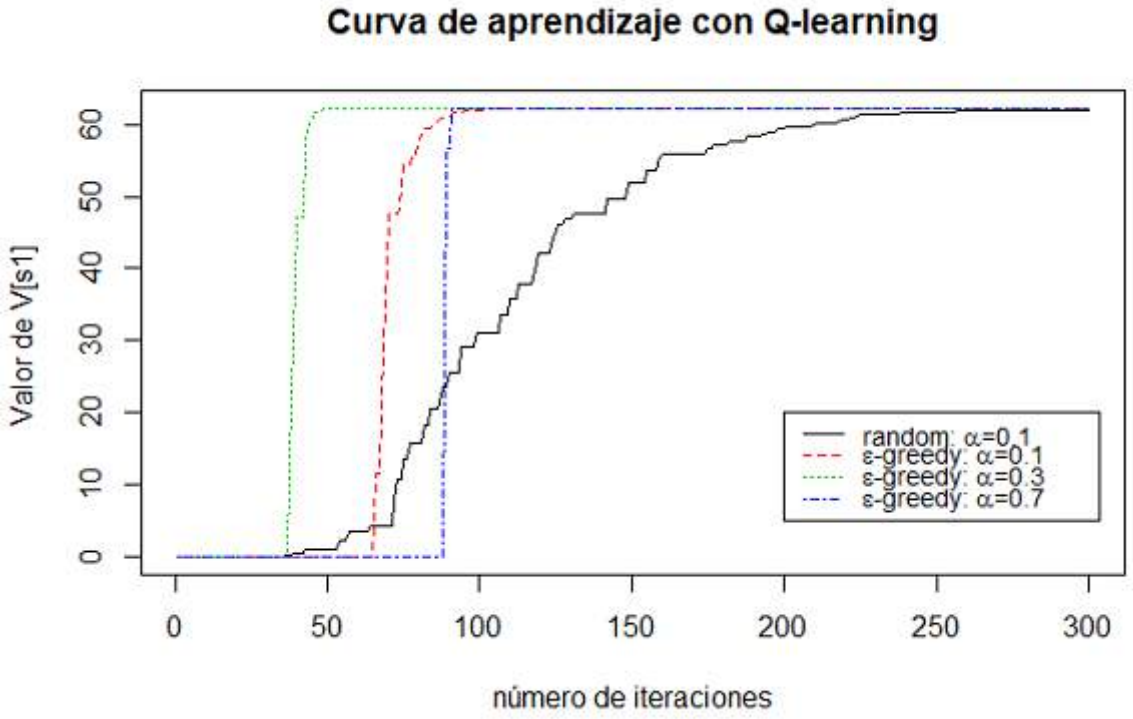

Figura 4.3: Comparación de la curva de aprendizaje según el valor de α.

El aprendizaje es mucho más directo con el aumento del valor del parámetro α, cuanto menor es este valor, más escalonada nos econtraremos la gráfica. Para este problema, observamos que el valor 0,3 es el mejor de los valores mostrados, negando así una correlación positiva entre el valor del parámetro y el aprendizaje.

4.3. *Wappo Game*: un ejemplo elaborado

En esta última sección vamos a estudiar ampliamente la aplicación del aprendizaje por refuerzo en *R* sobre el juego *Wappo Game*. El juego se puede adquirir en la página web de Google Play. Visitar el enlace `https://play.google.com/store/apps/details?id=com.pinkzero.wappogame&hl=es`.

El juego consta de 200 niveles. Nosotros desarrollaremos el nivel 1 simplificado. La pantalla de juego se presenta en la figura 4.4.

Figura 4.4: Pantalla inicial del juego Wappo Game

El jugador o agente mueve al personaje humano una casilla por turno y el objetivo es llegar a la casilla verde antes de que el enemigo nos alcance. Los muros no los puede atravesar ninguno de los personajes. La dinámica del juego es la siguiente: en primer lugar se mueve el humano, si no ha llegado a la casilla final se mueve el enemigo, y si éste no ha alcanzado al humano, vuelve a moverse el humano y se comienza de nuevo. El enemigo puede moverse hasta 2 casillas y la elección de movimientos viene programado automáticamente para que minimice la distancia entre su posición y la del humano, es decir, calcula las casillas de distancia y se mueve de manera que la distancia final vaya a ser menor. Para el cálculo no tiene en cuenta los muros. El movimiento del enemigo es diferente del juego original, ya que éste también intenta minimizar la distancia con el humano pero primero tratará de igualar la columna del humano y después lo hará con la fila.

El problema es levemente más complicado que el problema trabajado en la sección anterior, pero con pocas complejidades más en el problema, el coste de formulación y planteamiento aumenta notablemente. Por ejemplo, el estado del sistema ahora no depende solo de la casilla en la que se encuentra el agente, sino también depende del lugar del enemigo. Es decir, en este problema el estado tiene dos componentes: casilla del humano y casilla del enemigo. Observar que el estado inicial del juego es el $(s17 - s6)$. Cuando las dos componentes son iguales, el juego termina y el agente no ha conseguido el objetivo, y cuando la primera componente es igual a la casilla "H", el juego termina y el agente ha logrado completar la tarea.

Con estas simples nociones, programamos el juego en *R*, otorgando recompensas -1000 cuando pierde el agente, 1000 cuando gana y -1 por cada etapa que el humano se mueve y no finaliza el juego y 0 si no se mueve y no finaliza el juego.

Observar que si tratásemos este problema como un proceso de decisión de Markov, las dimensiones de porgramación son elevadas. Por ejemplo, deberíamos programar 4 matrices de transición de dimensión 25×25. Es por esta razón por la que trataremos este ejemplo solo desde la biblioteca *ReinforcementLearning*.

Comenzamos el estudio de este ejemplo procediendo de manera similar a la anterior. Al intentar obtener una gráfica de la curva de aprendizaje con los mismos parámetros que antes, obtenemos una gráfica constante en el valor 0, es decir, no aprende: el programa necesita más simulaciones del entorno para comenzar a aprender.

La recompensa 0 si no hay movimiento y -1 si lo hay, genera poco movimiento y por lo tanto, no acaba habiendo información en estados en los que la posición del humano es lejana a la inicial. Para solucionar este problema, vamos a simular muestras desde cada uno de los estados posibles. Observar que los estados posibles que nos interesan son $625 = 25 \cdot 25$. Las 26 casillas del humano menos la casilla "H" que no simulamos como inicial. Para el enemigo, son 26 casillas posibles menos en la que esté el

humano. Así pues, simulamos situaciones en cada una de estos 625 estados iniciales. En el siguiente script se muestra la realización de estas simulaciones.

```
out<-data.frame(State=NULL, Action=NULL, Reward=NULL, NexState=NULL,
           stringsAsFactors = FALSE)

for (i in 1:625) {
  for( ac in actions) {
    response<-WappoEnvironmment(wappo_states[i],ac)
    out<-rbind.data.frame(out,
                          data.frame(State = wappo_states[i], Action = ac,
                           Reward = response$Reward,
                           NextState = as.character(response$NextState) ,
                           stringsAsFactors = F))
   }
 }
```

Una vez simulado el entorno, vamos a tratar de solucionar el problema, es decir, de alcanzar una política óptima. Para ello aplicamos la función `ReinforcementLearning` a este conjunto de datos.

```
control <- list(alpha = 0.1, gamma = 0.9, epsilon = 0.1)
modelnonrd<-ReinforcementLearning(out, s = "State", a = "Action", r = "Reward",
                        s_new = "NextState", iter=50, control = control)
```

Entre los resultados obtenidos se encuentra la matriz Q que nos define la política a seguir. Presentamos a continuación una tabla con los valores de la matriz correspondientes al juego con el estado inicial $s17 - s6$.

State	Q				Next State
	down	left	right	up	
s17-s6	-0.99	-994.84	-1.86	**370.06**	s12-s11
s12-s11	-994.84	371.05	**543.95**	234.06	s13-s11
s13-s11	-870.58	381.91	**717.63**	381.91	s14-s11
s14-s11	-1.86	554.04	**868.59**	554.04	s15-s11
s15-s11	723.94	723.94	**994.84**	723.94	H-s11

Cuadro 4.1: Extracto de la matrix Q.

Por tanto, la política óptima que hemos obtenido es: *up - right - right - right - right*.

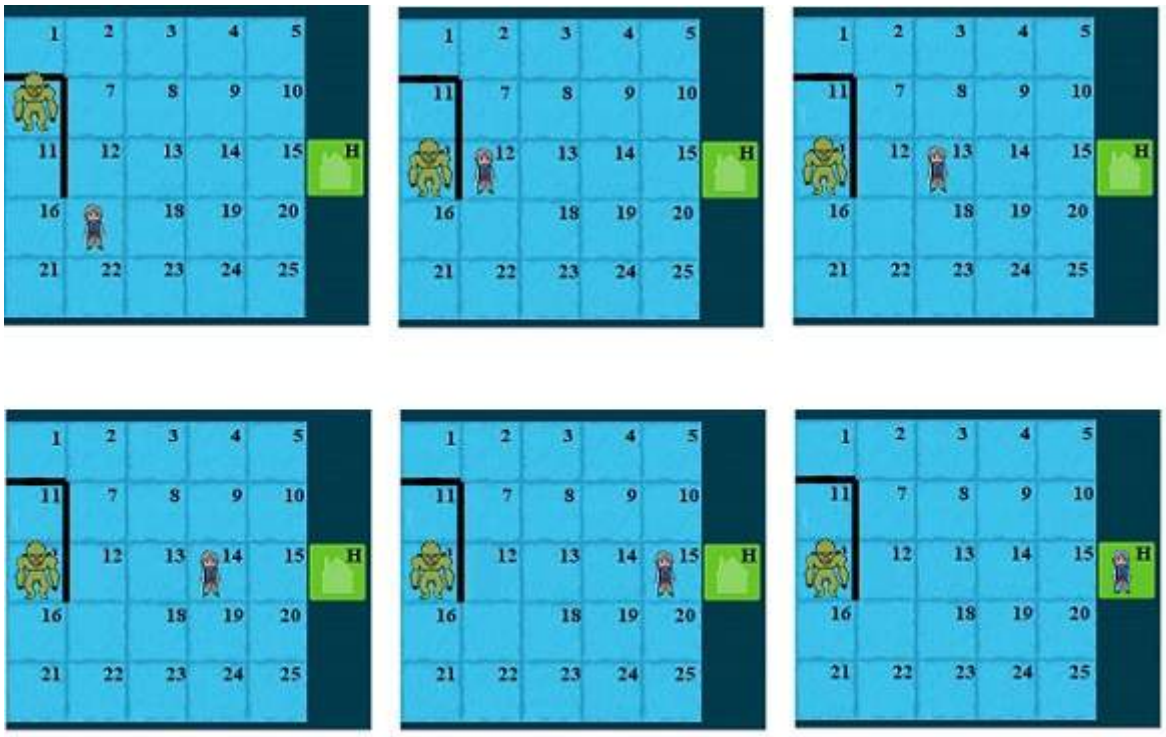

Figura 4.5: Wappo Game : SOLUCIÓN

Una vez solucionado el problema, nos preguntamos qué estados iniciales de todos los posibles, hacen posible el completar el juego. Para ello, utilizaremos los valores de la matriz Q. Recordar que los valores de $Q(s,a)$ representa el refuerzo esperado empezando desde el estado s y tomando la acción a. Tomar la acción que maximice esta Q para cada estado será la mejor opción y representará el refuerzo esperado desde el estado s y siguiendo la política óptima, es decir, el valor $V(s)$. Así pues, calculamos los valores de la función V como el máximo valor por filas de la matriz Q.

```
# calculate V function
V<- apply(modelnonrd$Q, 1, max)
```

Los valores obtenidos estarán comprendidos entre -1000 y 1000. Los valores positivos corresponderán a situaciones iniciales desde las que es posible finalizar el juego llegando a la casilla "H". Desde el resto de estados, no se puede alcanzar la solución del problema.

En nuestro caso donde el estado incial era $s17 - s6$, el valor obtenido 370,06 ha salido positivo, es decir, se puede solucionar el juego. Los estados en los que el humano se encuentre en la casilla contigua a la "H", $s15$, tendremos un valor de V cercano a 1000. Observando la tabla anterior 4.1 comprobamos esta deducción. En cambio, estados iniciales donde humano y enemigo estén muy próximos, se obtendrá un valor muy bajo en torno a -1000, ya que es muy probable que el enemigo acabe alcanzando al humano. Por ejemplo, el valor correspondiente al estado inicial $s8 - s3$ es $-994{,}84$.

El estado que posee el menor valor de V positivo es el estado $s1 - s6$, cuyo valor es 118,41. Este estado con el menor valor positivo de V se traduce en el juego como el estado que más etapas necesita para resolver el juego.

Otra lectura interesante son los estados con valores 0. Este resultado significa que desde ese estado lo más apropiado es ejecutar una acción que mantenga al humano en la misma casilla y así el enemigo

no se mueva tampoco. De esta manera no se resuelve el juego, pero tampoco se pierde. Por tanto, esta alternativa solo se da en estados donde la casilla correspondiente al humano limita con una de las paredes del juego. Por ejemplo, el estado $s24 - s22$, en el que ejecutraremos siempre la acción "*down*".

Análogamente, encontramos también estados con valores cercanos a -1, esto se debe a que nos estados desde los no se puede solucionar el problema pero se encuentran a un paso de estados pertenecientes a la situación anterior. Por ejemplo, desde el estado $s19 - s17$ la mejor opción es moverse hacia la derecha o hacia abajo y mantenerse ahí, ya que en cualquier otra alternativa el enemigo alcanza al humano.

Tras este problema podemos concluir que no solo el aprendizaje por refuerzo es una herramienta útil cuando desconocemos recompensas o probabilidades de transición, sino que también en problemas en los que el planteamiento del entorno tiene un coste elevado, estas funciones de simulación del entorno pueden ser necesarias. Además, estas herramientas no se limitan a la resolución del problema, sino que permiten estudiar a fondo diferentes alternativas y llegar a entender la estructura interna del problema. También señalar que aunque aumentando el número de iteraciones y la exploración en métodos como ε-greedy o softmax podríamos haber llegado a tener datos de todos los estados posibles, podemos hacer un primer muestreo de manera directa en cada uno de los estados como hemos hecho en esta ocasión, sobre todo en problemas donde el número de estados es tan elevado y llegar a cada uno de ellos es difícil.

El siguiente paso en el estudio de este ejemplo es admitir que el enemigo se comporte como otro agente y tenga movimiento propio. Ante este nuevo escenario, se nos plantearían muchas nuevas alternativas de aprendizaje. El problema principal es que hasta ahora, considerábamos el entorno como estacionario, pero en este nuevo caso no es posible, ya que el otro agente también está modificando el entorno. Esta nueva situación de aprendizaje se conoce como *aprendizaje multiagente*.

La gran diferencia de este nuevo escenario reside en el hecho de que todos y cada uno de los agentes producen algún efecto sobre el entorno y, por lo tanto, las acciones pueden tener resultados diferentes dependiendo del comportamiento del resto de agentes. Esta es precisamente la diferencia que plantea problemas a la hora de aplicar las técnicas de aprendizaje de refuerzo, ya que aparecen nuevos conceptos como el de cooperación entre agentes. En definitiva, debemos modificar los algoritmos y métodos a esta nueva situación. A pesar de encontrarse dentro del aprendizaje por refuerzo, este cambio de escenario implica una nueva área de estudio más complicada, pero que tiene su fundamento en la teoría descrita en este trabajo.

Bibliografía

[1] FERNÁNDEZ REBOLLO F., Aprendizaje por Refuerzo, Escuela Politécnica Superior, Univeridad Carlos III, Madrid, 2013. Disponible en http://ocw.uc3m.es/ingenieria-informatica/programacion-automatica-2013/paocwintroduccionrl.pdf.

[2] HUGHES, NEAL, Applying reinforcement learning to economic problems, Australian National University, 2014.

[3] MELO, FRANCISCO S, Convergence of Q-learning: A simple proof, Institute for Systems and Robotics, Tech. Rep, 2001, págs. 1-4. Disponible en http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

[4] MODI M, SAGAR C, REDDY V, VEETIL S, How to perform reinforcement learning with R. Material online disponible en https://dataaspirant.com/2018/02/05/reinforcement-learning-r/. Consultado 05-03-18.

[5] MOHRI M, Foundations of Machine Learning, Reinforcement learning. Material online disponible en https://cs.nyu.edu/~mohri/mls/ml_reinforcement_learning.pdf. Consultado 11-05-18.

[6] NETO, GONÇALO, From single-agent to multi-agent reinforcement learning: Foundational concepts and methods, Learning theory course, 2005. Disponible en http://www.cs.utah.edu/~tch/CS6380/resources/Neto-2005-RL-MAS-Tutorial.pdf.

[7] NOWÉ, ANN AND BRYS, TIM, A Gentle Introduction to Reinforcement Learning, International Conference on Scalable Uncertainty Management, Springer International Publishing, Suiza, 2016, págs. 18 - 32.

[8] PRÖLLOCHS, N AND FEUERRIEGEL, S, Reinforcement Learning, Business Analytics Practice, 2015. Disponible en http://www.is.uni-freiburg.de/ressourcen/business-analytics/13_reinforcementlearning.pdf.

[9] R CORE TEAM *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. http://www.R-project.org/.

[10] SILVER, DAVID AND HUANG, AJA AND MADDISON, CHRIS J AND GUEZ, ARTHUR AND SIFRE, LAURENT AND VAN DEN DRIESSCHE, GEORGE AND SCHRITTWIESER, JULIAN AND ANTONOGLOU, IOANNIS AND PANNEERSHELVAM, VEDA AND LANCTOT, MARC AND OTHERS, Mastering the game of Go with deep neural networks and tree search, nature, Nature Research, vol. 529, no. 7587, 2016, págs. 484 - 489. Disponible en http://web.iitd.ac.in/~sumeet/Silver16.pdf.

[11] SUTTON, RICHARD S AND BARTO, ANDREW G, *Reinforcement learning: An introduction*, MIT press Cambridge,1998. Disponible en http://incompleteideas.net/book/ebook/the-book.html.

[12] SZEPESVÁRI, CSABA, Algorithms for reinforcement learning, Synthesis lectures on artificial intelligence and machine learning, Morgan & Claypool Publishers, vol. 4, no. 1, 2010, págs. 1 - 103.

[13] UNIVERSITY OF WATERLOO, Sequential Decision Making and Reinforcement Learning. Material online disponible en `https://cs.uwaterloo.ca/~ppoupart/teaching/cs886-spring13/slides/cs886-module6-value-iteration.pdf`. Consultado 11-5-18.

[14] WATKINS, CHRISTOPHER JOHN CORNISH HELLABY, *Learning from delayed rewards*, King's College, Cambridge,1989.

Anexo 1

El contenido de este anexo está formado por las demostraciones de teoremas y proposiciones importantes comentados a lo largo del trabajo.

- *Demostración de la proposición 1.* La demostración de la desigualdad viene justificada por el siguiente desarrollo:

$$\begin{aligned} \| H^*(\widehat{\mathbf{V}}) - H^*(\mathbf{V}) \|_\infty &\leq \| \mathbf{R} + \gamma \mathbf{P}\widehat{\mathbf{V}} - \mathbf{R} - \gamma \mathbf{P}\mathbf{V} \|_\infty \\ &= \| \gamma \mathbf{P}(\widehat{\mathbf{V}} - \mathbf{V}) \|_\infty \\ &\leq \gamma \| \mathbf{P} \|_\infty \| \widehat{\mathbf{V}} - \mathbf{V} \|_\infty \\ &= \gamma \| \widehat{\mathbf{V}} - \mathbf{V} \|_\infty \end{aligned}$$

  Por tanto tenemos que $\| H^*(\widehat{\mathbf{V}}) - H^*(\mathbf{V}) \|_\infty \leq \gamma \| \widehat{\mathbf{V}} - \mathbf{V} \|_\infty$ que es lo que queríamos probar. □

- *Demostración del teorema 2.* Por definición $\mathbf{V}^* = H^{*(\infty)}(\mathbf{V}_0)$. En la proposición anterior hemos probado que H^* es un operador γ-contractible, es decir, $\| H^*(\widehat{\mathbf{V}}) - H^*(\mathbf{V}) \|_\infty \leq \gamma \| \widehat{\mathbf{V}} - \mathbf{V} \|_\infty$. En concreto también se tiene:

$$\| H^{*(n)}(\widehat{\mathbf{V}}) - H^{*(n)}(\mathbf{V}) \|_\infty \leq \gamma^n \| \widehat{\mathbf{V}} - \mathbf{V} \|_\infty \quad , \qquad \forall\, \mathbf{V}$$

  En concreto tenemos,

$$\| H^{*(n)}(\mathbf{V}^*) - H^{*(n)}(\mathbf{V}_0) \|_\infty \leq \gamma^n \| \mathbf{V}^* - \mathbf{V}_0 \|_\infty \xrightarrow[n\to\infty]{} 0$$

  Por tanto se tiene que $H^{*(\infty)}(\mathbf{V}_0) = \mathbf{V}^*$, $\forall\, \mathbf{V}_0$. Así pues se tiene que la convergencia se da sobre $\mathbf{V}^*$. La existencia y unicidad de un punto fijo para un operador contractible completa el resultado.

  □

- *Demostración del teorema 4.* Siendo π_{n+1} la política obtenida en la iteración n, por definición se tiene que:

$$\mathbf{R}_{\pi_{n+1}} + \gamma \mathbf{P}_{\pi_{n+1}} \mathbf{V}_n \geq \mathbf{R}_{\pi_n} + \gamma \mathbf{P}_{\pi_n} \mathbf{V}_n = \mathbf{V}_n$$

  De donde se tiene que $\mathbf{R}_{\pi_{n+1}} \geq (\mathbf{I} - \gamma \mathbf{P}_{\pi_{n+1}})\mathbf{V}_n$. Por lo tanto,

$$\mathbf{V}_{n+1} = (\mathbf{I} - \gamma \mathbf{P}_{\pi_{n+1}})^{-1} \mathbf{R}_{\pi_{n+1}} \geq \mathbf{V}_n$$

  □

  Antes de proceder a la demostración de las condiciones necesarias para la convergencia del algoritmo Q-learning, presentamos el siguiente lema.

  Lema 1. *Sea $\{\Delta_t\}$ un proceso aleatorio en $\mathbb{R}^n$ definido como sigue:*

$$\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \alpha_t(x) F_t(x)$$

  Este proceso converge a cero con probabilidad 1 si se dan la siguientes condiciones:

1. $0 \leq \alpha \leq 1$, $\sum_t \alpha_t = \infty \quad y \quad \sum_t \alpha_t^2 < \infty$
2. $||E\{F_t|\mathscr{F}_t\}||_W \leq \gamma||\Delta_t||_W$, *con* $\gamma < 1$
3. $\boldsymbol{var}\{F_t(s)|\mathscr{F}_t\} \leq C(1+||\Delta_t||_W^2)$, $C > 0$

- *Demostración del teorema 5.* Comenzamos la demostración reescribiendo la definición de $Q(s,a)$ en la iteración.

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)[r_t + \gamma \max_{b\in A} Q_t(s_{t+1},b)]$$

Tomamos ahora la función $\Delta_t(s,a)$ como

$$\Delta_t(s,a) = Q_t(s,a) - Q_t^*(s,a)$$

La idea de la demostración es probar que esta función Δ_t cumple las dos últimas propiedades del lema 1 (notar que la primera la cumple por hipótesis). De esta manera, se tendrá que Δ_t converge a cero con probabilidad 1 y por lo tanto, $Q_t(s,a)$ convergerá mediante la iteración Q-learning a $Q_t^*(s,a)$.

Definimos

$$F_t(s,a) = r(s,a,s') + \gamma \max_{b\in A} Q_t(s',b) - Q^*(s,a)$$

donde s' es el estado obtenido de aplicar la acción a en s. Así pues, veamos si se cumple la segunda condición:

$$E\{F_t(s,a)|\mathscr{F}_t\} = \sum_{s'\in S} T(s,a,s')[r(s,a,s') + \gamma \max_{b\in A} Q_t(s',b) - Q^*(s,a)] = (\mathbf{H}(Q_t(s,a)) - \mathbf{H}(Q^*(s,a))$$

siendo $\mathbf{H}$ el operador de Bellman. Utilizando la propiedad de que el operador tiene un punto fijo en $Q^*(s,a)$ y que es γ-contractible, se tiene directamente que

$$||E\{F_t(s,a)|\mathscr{F}_t\}||_\infty \leq \gamma||Q_t - Q^*||_\infty = \gamma||\Delta_t||_\infty$$

Luego se cumple la segunda condición.

Para probar que se cumple la tercera, veamos el siguiente desarrollo:

$$\begin{aligned}
\mathbf{var}\{F_t(s)|\mathscr{F}_t\} &= E\{(r(s,a,s') + \gamma \max_{b\in A} Q_t(s',b) - Q^*(s,a) - (\mathbf{H}(Q_t(s,a)) - Q^*(s,a)))^2\} \\
&= E\{(r(s,a,s') + \gamma \max_{b\in A} Q_t(s',b) - \mathbf{H}(Q_t(s,a)))^2\} \\
&= \mathbf{var}\{(r(s,a,s') + \gamma \max_{b\in A} Q_t(s',b)|\mathscr{F}_t\} \\
&\leq C(1+||\Delta_t||_\infty^2)
\end{aligned}$$

Esta última desigualdad se tiene ya que r es un valor acotado.

Así pues, también se cumple la tercera propiedad y por tanto, por el lema 1, Δ_t converge a cero con probabilidad 1, luego Q_t converge a Q^* con probabilidad 1. □

Anexo 2

En este anexo mostraremos los principales archivos de código en *R* que hemos utilizado durante el trabajo.

Funciones auxiliares

La siguiente función es una modificación de la función de *R* `sampleExperience`, necesaria para la implementación del método *softmax*.

```
adquiereExperiencia<- function (N, env, states, actions,
          actionSelection = "random", control = list(alpha = 0.1,
          gamma = 0.1, epsilon = 0.1), model = NULL, ...)
{
  if (!(N > 0 && length(N) == 1 && is.numeric(N) && floor(N) ==
        N)) {
    stop("Argument 'N' should be an integer > 0.")
  }
  if (!is.function(env)) {
    stop("Argument 'env' describing the environment must be of type function.")
  }
  if (!is.character(states)) {
    stop("Arguments 'states' must be of type 'character'.")
  }
  if (!is.character(actions)) {
    stop("Arguments 'actions' must be of type 'character'.")
  }
  if (class(model) != "rl" && !is.null(model)) {
    stop("Argument 'model' must be empty or of type 'rl'.")
  }
  if (!is.list(control)) {
    stop("Argument 'control' must be of type 'list'.")
  }
  if (is.null(control\$epsilon)) {
    stop("Missing or invalid control parameters.")
  }
  if (is.null(model)) {
    Q <- hash()
  }
  else {
    Q <- model\$Q_hash
  }
  for (i in unique(states)[!unique(states) {
```

```
    Q[[i]] <- hash(unique(actions), rep(0, length(unique(actions))))
  }
  #actionSelectionFunction <- lookupActionSelection(actionSelection)
  sampleStates <- sample(states, N, replace = TRUE)
  if(actionSelection=="random") {
  sampleActions <- sapply(sampleStates, randomActionSelection, Q=Q,
                                  epsilon=control\$epsilon)
  } else if ( actionSelection=="epsilon-greedy") {

    sampleActions <- sapply(sampleStates, epsilonGreedyActionSelection,
                            Q=Q, epsilon=control\$epsilon)

  } else if (actionSelection=="softmax") {
    sampleActions <- sapply(sampleStates, softmaxActionSelection,
                            Q=Q, tau=control\$tau)
  } else{
    stop("Rule for action selection not recognized. Corresponding argument
                has an invalid value.")
  }

  response <- lapply(1:length(sampleStates), function(x) env(sampleStates[x],
                                                             sampleActions[x]))
  response <- data.table::rbindlist(lapply(response, data.frame))
  out <- data.frame(State = sampleStates, Action = sampleActions,
                    Reward = response\$Reward,
                    NextState = as.character(response\$NextState),
                    stringsAsFactors = F)
  return(out)
}
```

Método *softmax*.

```
softmaxActionSelection<-function (Q, state, tau=1)
{
  posibles<- exp(values(Q[[state]])/tau)
  posibles<- posibles/sum(posibles)
  return(names(sample(values(Q[[state]]), 1, prob=posibles)))
}
```

Comandos para ejemplo Grid 3×4

Los comandos siguientes hacen referencia al planteamiento del problema, tanto para la librería de los MDP como a la librería *ReinforcementLearning*.

```
allstates<-as.vector(outer("s",1:11,FUN=paste, sep=""))
allactions<-c("up", "down", "left", "right")

nextstep<-array(NA, dim=c(length(allstates),length(allactions)))
```

```
rownames(nextstep)<-allstates
colnames(nextstep)<-allactions
nextstep["s11","up"]<-"s11"
nextstep["s11","down"]<-"s11"
nextstep["s11","left"]<-"s11"
nextstep["s11","right"]<-"s11"
nextstep["s6","up"]<-"s9"
nextstep["s6","down"]<-"s2"
nextstep["s6","left"]<-"s5"
nextstep["s6","right"]<-"s6"
nextstep["s7","up"]<-"s7"
nextstep["s7","down"]<-"s7"
nextstep["s7","left"]<-"s7"
nextstep["s7","right"]<-"s7"
nextstep["s8","up"]<-"s8"
nextstep["s8","down"]<-"s5"
nextstep["s8","left"]<-"s8"
nextstep["s8","right"]<-"s9"
nextstep["s9","up"]<-"s9"
nextstep["s9","down"]<-"s6"
nextstep["s9","left"]<-"s8"
nextstep["s9","right"]<-"s10"
nextstep["s10","up"]<-"s10"
nextstep["s10","down"]<-"s10"
nextstep["s10","left"]<-"s9"
nextstep["s10","right"]<-"s11"
nextstep["s1","up"]<-"s5"
nextstep["s1","down"]<-"s1"
nextstep["s1","left"]<-"s1"
nextstep["s1","right"]<-"s2"
nextstep["s2","up"]<-"s6"
nextstep["s2","down"]<-"s2"
nextstep["s2","left"]<-"s1"
nextstep["s2","right"]<-"s3"
nextstep["s3","up"]<-"s3"
nextstep["s3","down"]<-"s3"
nextstep["s3","left"]<-"s2"
nextstep["s3","right"]<-"s4"
nextstep["s4","up"]<-"s7"
nextstep["s4","down"]<-"s4"
nextstep["s4","left"]<-"s3"
nextstep["s4","right"]<-"s4"
nextstep["s5","up"]<-"s8"
nextstep["s5","down"]<-"s1"
nextstep["s5","left"]<-"s5"
nextstep["s5","right"]<-"s6"

reward<-rep(-1,length(allstates))
names(reward)<-allstates
reward["s7"]<- -100
reward["s11"]<- 100
```

```
tableau<-list()
tableau\$nextstep<-nextstep
tableau\$reward<-reward


## ReinforcementLearning

Pawn3x4Grid<-function(state, action, design=tableau) {
  nextstate<-tableau\$nextstep[toString(state),toString(action)]
  if (state ==nextstate) {
    reward<-0 # si  no me muevo de la celda no tengo recompensa
  } else
  {
    reward <- tableau\$reward[toString(nextstate)]
  }
  #reward <- tableau\$reward[toString(nextstate)] #prueba
  out<-list("NextState" = nextstate, "Reward" = reward)
  return(out)
}


## MDPtoolbox

## Definicion de matrices y de recompensas

  up<- matrix(c(0,0.1,0,0,0.8,0.1,0,0,0,0,0,0,
                0.1,0.1,0.1,0, 0.1, 0.6, 0, 0,0,0, 0,
                0, 0.1, 0.6, 0.1,0,0.1,0.1, 0, 0, 0, 0,
                0,0,0.1, 0.1, 0,0,0.8, 0, 0, 0, 0,
                0, 0, 0, 0, 0.1, 0.1,0, 0.7, 0.1, 0, 0,
                0, 0, 0, 0, 0.1, 0.1,0,0.1,0.6,0.1,0,
                0,0,0,0,0,0,1,0,0,0,0,
                0,0,0,0,0, 0, 0, 0.9, 0.1, 0.,0,
                0,0,0,0,0,0,0,.1, .8,.1,0,
                0,0,0,0,0,0,0,0,0,0.1,0.8,0.1,
                0,0,0,0,0,0,0,0,0,0,0,1),
    nrow=length(allstates), ncol=length(allstates), byrow=TRUE )

  up0<- matrix(c(0,0,0,0,1.0,0,0,0,0,0,0,0,
                0.0,0.0,0.0,0, 0.0, 1.0, 0, 0,0,0, 0,
                0, 0.0, 1.0, 0.0,0,0.0,0.0, 0, 0, 0, 0,
                0,0,0.0, 0.0, 0,0,1.0, 0, 0, 0, 0,
                0, 0, 0, 0, 0.0, 0.0,0, 1.0, 0, 0, 0,
                0, 0, 0, 0, 0, 0,0,0,1.0,0,0,
                0,0,0,0,0,0,1,0,0,0,0,
                0,0,0,0,0, 0, 0, 1, 0, 0.,0,
                0,0,0,0,0,0,0,0, 1.0,0,0,
                0,0,0,0,0,0,0,0,0,1.0,0,
                0,0,0,0,0,0,0,0,0,0,0,1),
              nrow=length(allstates), ncol=length(allstates), byrow=TRUE)
```

```
down<- matrix(c(0.9,0.1,0,0,0,0,0,0,0,0,0,
              0.1,0.8,0.1,0, 0, 0, 0, 0,0,0, 0,
              0, 0, 0.1, 0.8,0.1,0,0, 0, 0, 0, 0,
              0,0,0.1, 0.9, 0,0,0, 0, 0, 0, 0,
              0.7, 0.1, 0, 0, 0.1, 0.1,0, 0, 0, 0, 0,
              0, 0.8, 0, 0, 0.1, 0.1,0,0,0,0,0,
              0,0,0,0,0,0,1,0,0,0,0,
              0,0,0,0,0.8, 0, 0, 0.1, 0.1, 0.,0,
              0,0,0,0,0,0.8,0,0, .1,.1,0,
              0,0,0,0,0,0,0,0,0.1,0.8,0.1,
              0,0,0,0,0,0,0,0,0,0,1),
            nrow=length(allstates), ncol=length(allstates), byrow=TRUE)

down0<- matrix(c(1.0,0,0,0,0,0,0,0,0,0,0,
                0,1.0,0,0, 0, 0, 0, 0,0,0, 0,
                0, 0, 1.0, 0,0,0,0, 0, 0, 0, 0,
                0,0,0, 1.0, 0,0,0, 0, 0, 0, 0,
                1.0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0,
                0, 1.0, 0, 0, 0, 0,0,0,0,0,0,
                0,0,0,0,0,0,1,0,0,0,0,
                0,0,0,0,1.0, 0, 0, 0, 0, 0.,0,
                0,0,0,0,0,1.0,0,0, 0,0,0,
                0,0,0,0,0,0,0,0,0,1.0,0,
                0,0,0,0,0,0,0,0,0,0,1),
              nrow=length(allstates), ncol=length(allstates), byrow=TRUE )

left<- matrix(c(0.9,0.1,0,0,0,0,0,0,0,0,0,
                0.8,0.1,0.1,0, 0, 0, 0, 0,0,0, 0,
                0, 0.8, 0.1, 0.1,0,0,0, 0, 0, 0, 0,
                0,0,0.8,0.1, 0, 0,0.1,0, 0, 0, 0,
                0, 0, 0, 0, 0.9, 0.1,0, 0, 0, 0, 0,
                0, 0, 0, 0, 0.9, 0.1,0,0,0,0,0,
                0,0,0,0,0,0,1,0,0,0,0,
                0,0,0,0,0.1, 0, 0, 0.8, 0.1, 0.,0,
                0,0,0,0,0,0,0,0,0.8, .1,.1,0,
                0,0,0,0,0,0,0,0,0,0.8,0.1,0.1,
                0,0,0,0,0,0,0,0,0,0,0,1),
              nrow=length(allstates), ncol=length(allstates), byrow=TRUE )

left0<- matrix(c(1.0,0,0,0,0,0,0,0,0,0,0,0,
                1.0,0,0,0,0, 0, 0, 0, 0,0,0, 0,
                0, 1.0, 0, 0,0,0,0, 0, 0, 0, 0,
                0,0,1.0,0,0, 0,0,0, 0, 0, 0,
                0, 0, 0, 0, 1.0, 0,0, 0, 0, 0, 0,
                0, 0, 0, 0, 1.0, 0,0,0,0,0,0,
                0,0,0,0,0,0,1,0,0,0,0,
                0,0,0,0,0, 0, 0, 1.0, 0, 0,0,
                0,0,0,0,0,0,0,0,1.0, 0,0,0,
                0,0,0,0,0,0,0,0,0,1.0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,1),
```

```
              nrow=length(allstates), ncol=length(allstates), byrow=TRUE)

  right<- matrix(c(0.1,0.9,0,0,0,0,0,0,0,0,0,
                   0.1,0.1,0.8,0, 0, 0, 0, 0,0,0, 0,
                   0, 0.1, 0.1, 0.8,0,0,0, 0, 0, 0, 0,
                   0,0,0.1,0.9, 0, 0,0,0, 0, 0, 0,
                   0, 0, 0, 0, 0.9, 0.1,0, 0, 0, 0, 0,
                   0, 0, 0, 0, 0.9, 0.1,0,0,0,0,0,
                   0,0,0,0,0,0,1,0,0,0,0,
                   0,0,0,0,0.1, 0, 0, 0.1, 0.8, 0.,0,
                   0,0,0,0,0,0,0,0.1, .1,.8,0,
                   0,0,0,0,0,0,0,0,0,0.1,0.1,0.8,
                   0,0,0,0,0,0,0,0,0,0,0,1),
                 nrow=length(allstates), ncol=length(allstates), byrow=TRUE )

  right0<- matrix(c(0,1.0,0,0,0,0,0,0,0,0,0,
                    0,0,1.0,0,0, 0, 0, 0,0,0, 0,
                    0, 0, 0, 1.0,0,0,0, 0, 0, 0, 0,
                    0,0,0,1.0, 0, 0,0,0, 0, 0, 0,
                    0, 0, 0, 0, 0, 1.0,0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 1.0, 0,0,0,0,0,
                    0,0,0,0,0,0,1,0,0,0,0,
                    0,0,0,0,0, 0, 0, 0, 1.0, 0,0,
                    0,0,0,0,0,0,0,0,0, 0,1.0,0,
                    0,0,0,0,0,0,0,0,0,0,0,1.0,
                    0,0,0,0,0,0,0,0,0,0,0,1),
                  nrow=length(allstates), ncol=length(allstates), byrow=TRUE )

    T <-   list(up=up, left=left,
                down=down, right=right)
    T0 <- list(up=up0, left=left0,          #modelo sin errores
            down=down0, right=right0)

    Rup<-matrix(rep(0,121),
                nrow=length(allstates), ncol=length(allstates), byrow=TRUE
    )
    Rup[1,5]<--1
    Rup[2,6]<--1
    Rup[4,7]<--100
    Rup[5,8]<--1
    Rup[6,9]<--1

    Rdown<-matrix(rep(0,121),
                nrow=length(allstates), ncol=length(allstates), byrow=TRUE
    )
    Rdown[5,1]<--1
    Rdown[6,2]<--1
    Rdown[7,4]<-0
    Rdown[8,5]<--1
    Rdown[9,6]<--1
```

```
    Rleft<-matrix(rep(0,121),
                  nrow=length(allstates), ncol=length(allstates), byrow=TRUE
    )
    Rleft[2,1]<--1
    Rleft[3,2]<--1
    Rleft[4,3]<--1
    Rleft[6,5]<--1
    Rleft[9,8]<--1
    Rleft[10,9]<--1

    Rright<-matrix(rep(0,121),
                  nrow=length(allstates), ncol=length(allstates), byrow=TRUE
    )
    Rright[1,2]<--1
    Rright[2,3]<--1
    Rright[3,4]<--1
    Rright[5,6]<--1
    Rright[8,9]<--1
    Rright[9,10]<--1
    Rright[10,11]<-100

    R0<-array(NA,dim=c(11,11,4))
    R0[,,1]<-Rup
    R0[,,2]<-Rleft
    R0[,,3]<-Rdown
    R0[,,4]<-Rright


  require(MDPtoolbox)
  mdp_check(T, R)
  mdp_check(T0, R0)

  mdp_computePR(T,R)
  mdp_computePR(T,R0)
  mdp_computePR(T0,R0)

  m <- mdp_policy_iteration(P=T, R=R0, discount=0.9)
  m
  m0 <- mdp_policy_iteration(P=T0, R=R0, discount=0.9)
  m0
  m0<- mdp_value_iteration(T0,R0, discount=0.9)
  m0
```

A continuación se presenta el script correspondiente a la realización de la gráfica 4.2.

```
allstates<-as.vector(outer("s",1:11,FUN=paste, sep=""))
actions<- c("down", "left", "right","up")

require(ReinforcementLearning)
control <- list(alpha = 0.1, gamma = 0.9, epsilon = 0.1)
```

```
set.seed(1214)
data<-NULL
data <- sampleExperience(N = 100, env = Pawn3x4Grid,
                         states = allstates, actions = actions,
                         design=tableau)
head(data)

model0<-NULL
model0 <- ReinforcementLearning(data, s = "State", a = "Action", r = "Reward",
                                s_new = "NextState", iter=1, control = control)

modelrd<-NULL
modeleg<-NULL
modelsm<-NULL

control<- list(alpha = 0.1, gamma = 0.9, epsilon = 0.1)
controlsm <- list(alpha = 0.1, gamma = 0.9, epsilon = 0.1, tau=100)

nrep<-300
V<-array(0, dim=c(nrep,3))
Vrd<-rep(0,nrep)
Veg<-rep(0,nrep)
Vsm<-rep(0,nrep)

model0<-NULL
model0 <- ReinforcementLearning(data, s = "State", a = "Action", r = "Reward",
                                 s_new = "NextState", iter=1, control = control)
 for (i in (1:nrep)){
 new_data<-NULL
 new_data <- sampleExperience(N = 10,
                              env = Pawn3x4Grid,
                              design=tableau,
                              states = allstates,
                              actions = actions,
                              model = model0,
                              actionSelection = "random",
                              control=control)

 modelrd <- ReinforcementLearning(new_data,
                                  s = "State",
                                  a = "Action",
                                  r = "Reward",
                                  s_new = "NextState",
                                  iter=2, control = control,
                                  model=model0)
 model0<-modelrd
 Vrd[i]<-max(model0\$Q["s1",])
 }

model0<-NULL
```

```
model0 <- ReinforcementLearning(data, s = "State", a = "Action", r = "Reward",
                                s_new = "NextState", iter=1, control = control)
for (i in (1:nrep)){
 new_data<-NULL
 new_data <- sampleExperience(N = 10,
                              env = Pawn3x4Grid,
                              design=tableau,
                              states = allstates,
                              actions = actions,
                              model = model0,
                              actionSelection = "epsilon-greedy",
                              control=control)
 modeleg <- ReinforcementLearning(new_data,
                               s = "State",
                               a = "Action",
                               r = "Reward",
                               s_new = "NextState",
                               iter=2, control = control,
                               model=model0)
  model0<-modeleg
  Veg[i]<-max(model0\$Q["s1",])
}

model0<-NULL
model0 <- ReinforcementLearning(data, s = "State", a = "Action", r = "Reward",
                                s_new = "NextState", iter=1, control = control)
  for (i in (1:nrep)){
  new_data<-NULL
  new_data <- adquiereExperiencia(N = 10,
                               env = Pawn3x4Grid,
                               design=tableau,
                               states = allstates,
                               actions = actions,
                               model = model0,
                               actionSelection = "softmax",
                               control=controlsm)
  modelsm <- ReinforcementLearning(new_data,
                               s = "State",
                               a = "Action",
                               r = "Reward",
                               s_new = "NextState",
                               iter=2, control = controlsm,
                               model=model0)
  model0<-modelsm
  Vsm[i]<-max(model0\$Q["s1",])
}

V[,1]<-Vrd
V[,2]<-Veg
V[,3]<-Vsm
matplot(V, type="n",main="Curva de aprendizaje con Q-learning",
```

```
        xlab="número de iteraciones", ylab="Valor de V[s1]")
matlines(V)
legend(200,20,c("random", expression(paste(epsilon, "-greedy")), "softmax"),
                cex=.7, col=1:3,lty=1:3)
```

De manera similar se construye la gráfica 4.3.

Comandos para ejemplo Wappo Game

Vamos a exponer el script donde se recoge las funciones necesarias para el funcionamiento del juego Wappo Game.

```
WappoEnvironmment<- function(state, action){

  next_state<- state
  ## obtain human agent position
  ## obtain enemy agent position
  h_state<-toString(read.table(text=state, sep="-", as.is=TRUE)$V1)
  e_state<-toString(read.table(text=state, sep="-", as.is=TRUE)$V2)

   h_new_state<-obtain_h_pos(h_state,action)
   e_new_state<-e_state

  if (h_new_state == h_state){
    reward<-0
  } else if (h_new_state == toString("H") ) {
    reward <- 1000
      }
  else {
    reward<- -1
    e_new_state<-NULL
   e_new_state<-wappo_fetch(h_new_state,e_state)
    if (h_new_state == e_new_state) reward <- -1000
  }

  # build new state
  next_state <-paste(h_new_state,e_new_state, sep="-")

   out<-list("NextState" = next_state, "Reward" = reward)
   return(out)
}


obtain_h_pos<-function(state, action){

  h_allstates<-c(as.vector(outer("s",1:25,FUN=paste, sep="")),"H")
  h_allactions<-c("up", "down", "left", "right")
  hnextst<-array(NA, dim=c(length(h_allstates),length(h_allactions)))
```

```
  rownames(hnextst)<-h_allstates
  colnames(hnextst)<-h_allactions

hnextst["s1","up"]<- toString("s1")
hnextst["s1","left"]<- toString("s1")
hnextst["s1","down"]<- toString("s1") #wall down
hnextst["s1","right"]<- toString("s2")
hnextst["s2","up"]<- toString("s2")
hnextst["s2","left"]<- toString("s1")
hnextst["s2","down"]<- toString("s7")
hnextst["s2","right"]<- toString("s3")
hnextst["s3","up"]<- toString("s3")
hnextst["s3","left"]<- toString("s2")
hnextst["s3","down"]<- toString("s8")
hnextst["s3","right"]<- toString("s4")
hnextst["s4","up"]<- toString("s4")
hnextst["s4","left"]<- toString("s3")
hnextst["s4","down"]<- toString("s9")
hnextst["s4","right"]<- toString("s5")
hnextst["s5","up"]<- toString("s5")
hnextst["s5","left"]<- toString("s4")
hnextst["s5","down"]<- toString("s10")
hnextst["s5","right"]<- toString("s5")
hnextst["s6","up"]<- toString("s6") # wall up
hnextst["s6","left"]<- toString("s6")
hnextst["s6","down"]<- toString("s11")
hnextst["s6","right"]<- toString("s6") # wall right
hnextst["s7","up"]<- toString("s2")
hnextst["s7","left"]<- toString("s7") # wall left
hnextst["s7","down"]<- toString("s12")
hnextst["s7","right"]<- toString("s8")
hnextst["s8","up"]<- toString("s3")
hnextst["s8","left"]<- toString("s7")
hnextst["s8","down"]<- toString("s13")
hnextst["s8","right"]<- toString("s9")
hnextst["s9","up"]<- toString("s4")
hnextst["s9","left"]<- toString("s8")
hnextst["s9","down"]<- toString("s14")
hnextst["s9","right"]<- toString("s10")
hnextst["s10","up"]<- toString("s5")
hnextst["s10","left"]<- toString("s9")
hnextst["s10","down"]<- toString("s15")
hnextst["s10","right"]<- toString("s10")
hnextst["s11","up"]<- toString("s6")
hnextst["s11","left"]<- toString("s11")
hnextst["s11","down"]<- toString("s16")
hnextst["s11","right"]<- toString("s11") #wall right
hnextst["s12","up"]<- toString("s7")
hnextst["s12","left"]<- toString("s12") #wall left
hnextst["s12","down"]<- toString("s17")
hnextst["s12","right"]<- toString("s13")
```

```
hnextst["s13","up"]<- toString("s8")
hnextst["s13","left"]<- toString("s12")
hnextst["s13","down"]<- toString("s18")
hnextst["s13","right"]<- toString("s14")
hnextst["s14","up"]<- toString("s9")
hnextst["s14","left"]<- toString("s13")
hnextst["s14","down"]<- toString("s19")
hnextst["s14","right"]<- toString("s15")
hnextst["s15","up"]<- toString("s10")
hnextst["s15","left"]<- toString("s14")
hnextst["s15","down"]<- toString("s20")
hnextst["s15","right"]<- toString("H") # "Home-Goal"
hnextst["s16","up"]<- toString("s11")
hnextst["s16","left"]<- toString("s16")
hnextst["s16","down"]<- toString("s21")
hnextst["s16","right"]<- toString("s17")
hnextst["s17","up"]<- toString("s12")
hnextst["s17","left"]<- toString("s16")
hnextst["s17","down"]<- toString("s22")
hnextst["s17","right"]<- toString("s18")
hnextst["s18","up"]<- toString("s13")
hnextst["s18","left"]<- toString("s17")
hnextst["s18","down"]<- toString("s23")
hnextst["s18","right"]<- toString("s19")
hnextst["s19","up"]<- toString("s14")
hnextst["s19","left"]<- toString("s18")
hnextst["s19","down"]<- toString("s24")
hnextst["s19","right"]<- toString("s20")
hnextst["s20","up"]<- toString("s15")
hnextst["s20","left"]<- toString("s19")
hnextst["s20","down"]<- toString("s25")
hnextst["s20","right"]<- toString("s20")
hnextst["s21","up"]<- toString("s16")
hnextst["s21","left"]<- toString("s21")
hnextst["s21","down"]<- toString("s21")
hnextst["s21","right"]<- toString("s22")
hnextst["s22","up"]<- toString("s17")
hnextst["s22","left"]<- toString("s21")
hnextst["s22","down"]<- toString("s22")
hnextst["s22","right"]<- toString("s23")
hnextst["s23","up"]<- toString("s18")
hnextst["s23","left"]<- toString("s22")
hnextst["s23","down"]<- toString("s23")
hnextst["s23","right"]<- toString("s24")
hnextst["s24","up"]<- toString("s19")
hnextst["s24","left"]<- toString("s23")
hnextst["s24","down"]<- toString("s24")
hnextst["s24","right"]<- toString("s25")
hnextst["s25","up"]<- toString("s20")
hnextst["s25","left"]<- toString("s24")
hnextst["s25","down"]<- toString("s25")
```

```
hnextst["s25","right"]<- toString("s25")
hnextst["H","up"]<- toString("H")
hnextst["H","left"]<- toString("H")
hnextst["H","down"]<- toString("H")
hnextst["H","right"]<- toString("H")

new_state<-hnextst[toString(state),toString(action)]
return(new_state)
}


evilsteps<- list()

evilsteps[[1]]<-c("s2","s3","s1","s7")
evilsteps[[2]]<-c("s1","s2","s3","s4","s7","s12","s8")
evilsteps[[3]]<-c("s1","s2","s3","s4","s5","s7","s8","s9","s13")
evilsteps[[4]]<-c("s2","s3","s4","s5","s8","s9","s10","s14")
evilsteps[[5]]<-c("s3","s4","s5","s9","s10","s15")
evilsteps[[6]]<-c("s6","s11","s16")
evilsteps[[7]]<-c("s7","s8","s9","s12","s13","s1","s2","s3","s17")
evilsteps[[8]]<- c("s2","s3","s4","s7","s8","s9","s10","s12","s13","s14","s18")
evilsteps[[9]]<-c("s3","s4","s5","s7","s8","s9","s10","s13","s14","s15","s19")
evilsteps[[10]]<-c("s4","s5","s8","s9","s10","s14","s15","s20")
evilsteps[[11]]<-c("s6","s11","s16","s17")
evilsteps[[12]]<-c("s2","s7","s8","s12","s13","s14","s16","s17","s18","s22")
evilsteps[[13]]<-c("s3","s7","s8","s9","s12","s13","s14","s15","s17",
                   "s18","s19","s23")
evilsteps[[14]]<-c("s4","s8","s9","s10","s12","s13","s14","s15","s18",
                    "s19","s20","s24")
evilsteps[[15]]<-c("s5","s9","s10","s13","s14","s15","s19","s20","s25")
evilsteps[[16]]<-c("s6","s11","s12","s16","s17","s18","s21","s22")
evilsteps[[17]]<-c("s7","s11","s12","s13","s16","s17","s18","s19","s21",
                   "s22","s23")
evilsteps[[18]]<-c("s8","s12","s13","s14","s16","s17","s18","s19","s20",
                    "s22","s23","s24")
evilsteps[[19]]<-c("s9","s13","s14","s15","s17","s18","s19","s20","s23",
                    "s24","s25")
evilsteps[[20]]<-c("s10","s14","s15","s18","s19","s20","s23","s24","s25")
evilsteps[[21]]<-c("s11","s16","s17","s21","s22","s23")
evilsteps[[22]]<-c("s12","s16","s17","s18","s21","s22","s23","s24")
evilsteps[[23]]<-c("s13","s17","s18","s19","s21","s22","s23","s24","s25")
evilsteps[[24]]<-c("s14","s18","s19","s20","s22","s23","s24","s25")
evilsteps[[25]]<-c("s15","s19","s20","s23","s24","s25")


st2num<-function(x) return(as.integer(strsplit(x,split="s")[[1]][2]))

#dist_grid<-matrix(0,nrow=25, ncol=25)
a<-matrix(c(0,1,2,3,4,1,0,1,2,3,2,1,0,1,2,3,2,1,0,1,4,3,2,1,0), ncol=5,byrow=TRUE)
unos<-matrix(rep(1,25), ncol=5)
```

```
dist_grid<-kronecker(unos,a)+kronecker(a,unos)

how_dist<-function(ih,ie_a)return( dist_grid[ih,ie_a])

wappo_fetch <- function(hst,est){
  ih<-st2num(hst)
  ie<-st2num(est)
  ie_fetch<-sapply(evilsteps[[ie]],FUN=st2num)
  dist_e<-sapply(ie_fetch,FUN=how_dist, ih=ih)
  i_mindist<-order(dist_e)[1]
  e_new_st<-evilsteps[[ie]][i_mindist]
  return(e_new_st)
}
```

Los comandos siguientes corresponden al estudio del juego y búsqueda de soluciones.

```
h_allstates<-c(as.vector(outer("s",1:25,FUN=paste, sep="")),"H")
e_allstates<-as.vector(outer("s",1:25,FUN=paste, sep=""))
not_valid_states<-diag(as.matrix(outer(h_allstates[-26],e_allstates,FUN=paste,
                                  sep="-"), nrow=25))

wappo_states<-as.vector(outer(h_allstates,e_allstates,FUN=paste, sep="-"))
actions<- c("up", "down", "left", "right")

wappo_states<-setdiff(wappo_states,not_valid_states)

control <- list(alpha = 0.1, gamma = 0.9, epsilon = 0.1)

require(ReinforcementLearning)

out<-data.frame(State=NULL, Action=NULL, Reward=NULL, NexState=NULL,
                              stringsAsFactors = FALSE)

for (i in 1:625) {
  for( ac in actions) {
    response<-WappoEnvironmment(wappo_states[i],ac)
    out<-rbind.data.frame(out,
                          data.frame(State = wappo_states[i], Action = ac,
                           Reward = response$Reward,
                           NextState = as.character(response$NextState) ,
                           stringsAsFactors = F)
    )
  }}

control <- list(alpha = 0.1, gamma = 0.9, epsilon = 0.1)
modelnonrd<-ReinforcementLearning(out, s = "State", a = "Action", r = "Reward",
                      s_new = "NextState", iter=50, control = control)

# calculate V function
```

```
V<- apply(modelnonrd$Q, 1, max)
sort(V) #Ordena los valores de V

sum((V>0) ) #Cuenta el numero de V positivas
sum((V<0) )
sum((V==0) )

#RESULTADOS
#> values(modelnonrd$Q_hash[["s17-s6"]])
#        down         left        right           up
#  -0.9948462 -994.8462248   -1.8644390  370.0610569
#> values(modelnonrd$Q_hash[["s12-s11"]])
#     down      left     right        up
#-994.8462  371.0559  543.9518  234.0664
#> values(modelnonrd$Q_hash[["s13-s11"]])
#     down      left     right        up
#-870.5876  381.9111  717.6353  381.9110
#> values(modelnonrd$Q_hash[["s14-s11"]])
#      down       left      right         up
# -1.864439 554.044318 868.597880 554.044566
#> values(modelnonrd$Q_hash[["s15-s11"]])
#    down     left    right       up
#723.9486 723.9461 994.8462 723.9461
```

A partir de estos últimos resultados, hemos formado la tabla 4.1.