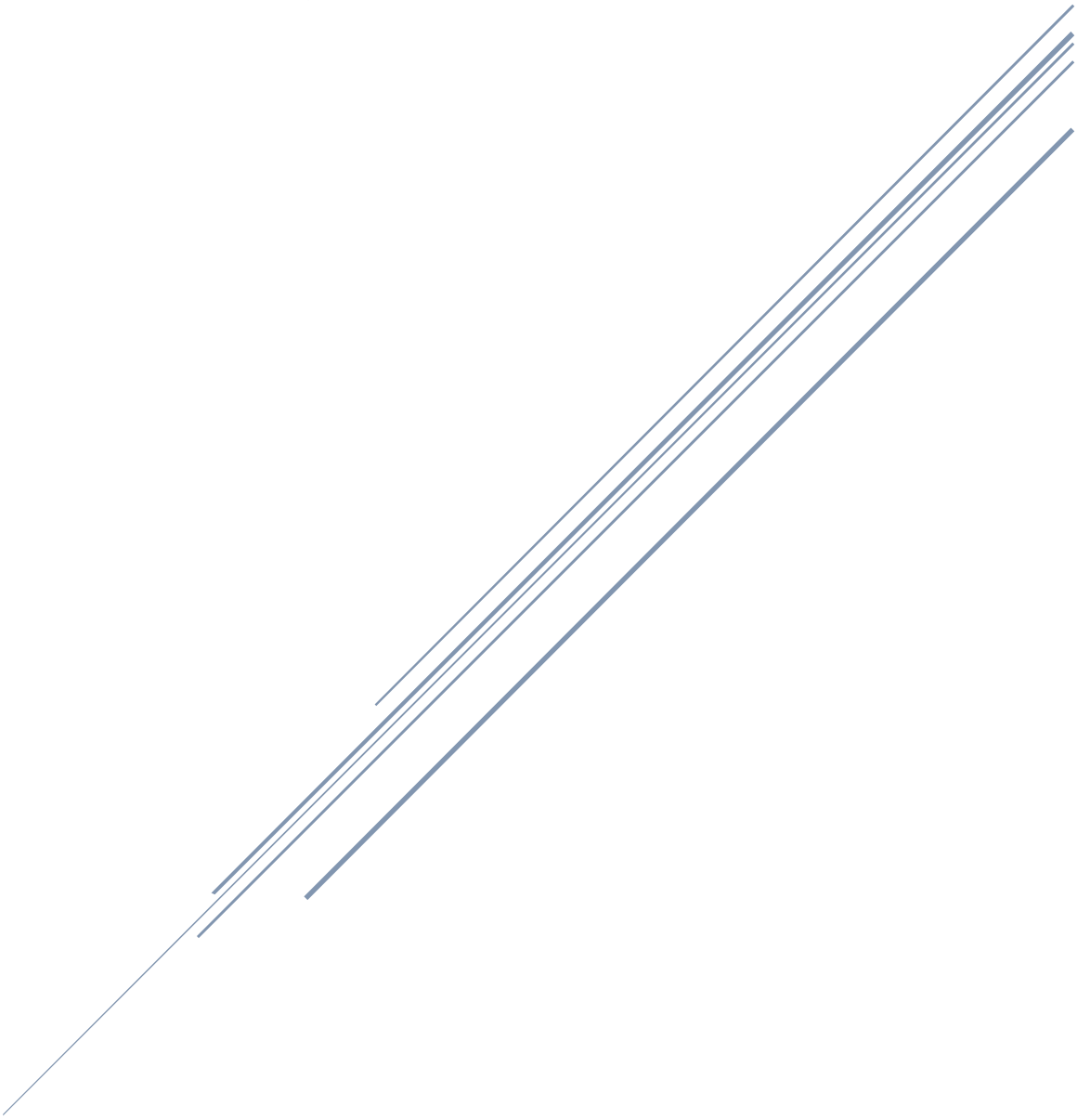
APRENDIZAJE AUTOMÁTICO

Práctica 2

Universidad Carlos III Madrid, Ingeniería Informática



IVAN AGUADO PERULERO – 100405871
JORGE SERRANO PÉREZ – 100405987

Contenido

1.- INTRODUCCIÓN

El objetivo de esta práctica es construir un agente en el dominio del juego Pac-Man que funcione de forma automática mediante el algoritmo *Q-learning*. Para ello se utilizarán técnicas de aprendizaje por refuerzo. Se parte de un código inicial del juego, al que tendremos que añadir el agente en cuestión.

Este documento se divide en distintas fases. En primer lugar, la fase en la que se seleccionará la función de refuerzo y la información de cada estado. En segundo lugar, se procederá a implementar el agente en el código. Por último, se procederá a evaluar el comportamiento del agente en distintos mapas y con distintas configuraciones.

2.- FASE 1. SELECCIÓN DE LA INFORMACIÓN DEL ESTADO Y FUNCIÓN DE REFUERZO

En esta fase se especificarán y justificarán los distintos atributos que se han elegido para representar el estado del juego. Se describirá también la función de refuerzo final empleada, así como el posible tratamiento de datos que se lleve a cabo.

ELECCIÓN DE ATRIBUTOS

En el agente se van a emplear tuplas de experiencia del tipo (*estado_tick, acción*, *estado_tick_siguiente*, *refuerzo*). En el *estado_tick* y el *estado_tick_siguiente* se han utilizado distintos atributos para representarlos. Los atributos que hemos elegido nosotros se corresponden con la posición relativa del fantasma más cercano respecto a Pac-Man. Por ello, los posibles valores pueden ser que el fantasma se encuentre al norte de su posición, al sur, al este o al oeste. Es por eso que el número total es cuatro, y se corresponde con el número total de filas de la *tabla Q*. Hemos elegido estos atributos por distintos motivos. El primero es que nos proporciona información genérica que no está ligada a ningún laberinto en particular. De esta manera, el Pac-man no se centra solo en un mapa y resuelve la mayoría de los mismos. Por otro lado, al tener menos atributos, el aprendizaje será más rápido. Finalmente, se trata también de atributos nominales que solo pueden tomar los valores "*North*", "*South*", "*East*" y "*West*". Esto también nos beneficia porque hace que la *tabla Q* sea más pequeña, que al final es en lo que se

basa el aprendizaje por refuerzo mediante Q-learning (actualizar la tabla en la ejecución de cada acción).

Destacar, sin embargo, que se han incluido ciertos aspectos en la elección de estos estados que mejoran en gran medida la resolución en distintos mapas. Estos son el incluir de manera combinada si en las casillas de alrededor del Pac-man existen paredes que dificulten el terminar la partida. Al combinarlas, no se aumenta el tamaño de la tabla, pero ayuda enormemente a generalizar. Por otro lado, también se añade que, en caso de pared, el Pac-man elija la mejor opción, de nuevo en base a la posición relativa respecto al fantasma.

Pasaron por nuestra cabeza otros atributos que podrían ayudar a mejorar el proceso de aprendizaje. Entre ellos podemos mencionar la combinación de la posición relativa junto con la distancia al Pac-Man más cercano. Sin embargo, descartamos esta idea rápidamente porque aumentaba mucho el tamaño de la *tabla Q* al existir tantas combinaciones. Por otro lado, pensamos también el incluir las posiciones de los fantasmas, pero por el mismo motivo también lo desechamos. Finalmente, se nos ocurrió el añadir si en las casillas de alrededor del Pac-man existían paredes o no, pero de manera independiente. Fue ahí cuando nos dimos cuenta de que podíamos combinar ambos atributos y de esta manera simplificar la tabla. Por otra parte, una idea que también parecía buena en un primer momento era la de crear ocho estados en vez de cuatro añadiendo las posiciones relativas noreste, noroeste, sudeste y sudoeste. Al implementarla, la mejora no era suficiente y no compensaba al aumentar el tamaño de la tabla, por lo que tampoco la tuvimos en cuenta.

ACCIONES POSIBLES

Describiremos ahora brevemente las distintas acciones posibles en la tupla de experiencia. Estas son tan solo realizar movimientos de tamaño uno en las cuatro direcciones posibles, es decir, "*North*", "*South*", "*East*" y "*West*".

FUNCIÓN DE REFUERZO

Por último, se va a exponer y explicar la función de refuerzo seleccionada finalmente, que se relaciona con acortar distancia:

> *R(S, a): { 1 si a = S, 0 en otro caso }* siendo *S*, el estado actual y *a* la acción ejecutada en desde ese estado.

Como se ha explicado anteriormente el estado cuenta con un atributo que define la posición relativa del fantasma más cercano respecto al mismo. De esta manera, si la acción a realizar desde un estado coincide con la acción que le acerca a su objetivo, ese movimiento se reforzará positivamente.

TRATAMIENTO DE DATOS

Para conseguir el atributo que necesita nuestro estado se trabajará sobre dos datos principales, la posición del Pac-man y la posición de los fantasmas. Con esta información podremos saber dónde se encuentra el fantasma más cercano respecto a nuestro agente. Otro dato auxiliar, aunque no menos importante, es la situación de las paredes adyacentes en cuanto a la posición del Pac-man. Esta también influirá a la hora de elegir el estado, pues aunque la posición relativa del fantasma sea una determinada, si en la "casilla" contigua hacia él hay una pared, esta hará que el estado varíe. Se profundizará más en cómo se tratan estos datos en la siguiente fase.

3.- FASE 2. CONSTRUCCIÓN DEL AGENTE

En esta segunda fase se describe el código generado para llevar a cabo el aprendizaje de la política de comportamiento, además del agente final implementado. Se realizará un histórico sobre este último, describiendo las diferencias frente a agentes pasados.

IMPLEMENTACIÓN DEL CÓDIGO GENERADO

En primer lugar, se va a describir el código generado para el agente final implementado. Partimos de la clase *QLearningAgent* del tutorial 4 que hicimos previamente para familiarizarnos con el aprendizaje por refuerzo. Por ello, el primer paso es copiarla al fichero *bustersAgents.py* donde se encuentran el resto de posibles agentes. Tendremos que cambiar el agente del que hereda, ya que en este caso no existe *ReinfrocementAgent*. Siguiendo la estructura del resto de agentes, tendremos el método *registerInitialState* donde asignaremos los valores de las distintas variables de instancia, como pueden ser epsilon, alpha y el factor de descuento.

Epsilon determina la probabilidad de ejecutar una acción aleatoria o seguir la de mejor política. Debido a esto el valor que le asignamos es pequeño, 0.05, pues nos interesa que el fantasma siga casi siempre el mejor camino en función de su experiencia. Sin embargo, también es cierto que en fases iniciales el valor de esta variable era más alto, facilitando así la exploración del agente, con lo que conseguirá encontrar posibles mejores políticas y desbloquearse si entrase en un bucle. En cuanto al factor de descuento, este establece un balance entre el refuerzo

inmediato y el refuerzo a largo plazo. En nuestro caso utilizamos un valor alto, 0.8, ya que el Pac-man no es un juego que se gane en un solo turno ejecutando una sola acción, por lo que es importante tener en cuenta los refuerzos futuros. Por último, alpha, es decir la tasa de aprendizaje, comenzó con un valor de 0.8. De esta manera la rapidez con la que se modifican las estimaciones comienza siendo alta al no dar casi peso al valor actual de la tabla. Según se iba avanzando, esta se fue reduciendo a medida que continuaba el juego y aprendía el agente.

Volviendo al método *registerInitialState* también hemos inventado una funcionalidad que detecta si existe un fichero *qtable.txt*, que será desde el cual leerá y actualizará la *tabla Q* del algoritmo Q-learning. Si no existe, haremos uso del método *initializeQTable*, que recibe el número de filas de la tabla para inicializar todos sus valores a 0 haciendo uso de la biblioteca *numpy*.

A continuación, en cada tick de juego habrá que construir la tupla de experiencia previamente mencionada, utilizando los atributos previamente seleccionados para los estados. Esta tupla se pasará al método *update*, que es muy similar al del tutorial 4. Esta se encarga de actualizar la *tabla Q* en memoria, y será el método *writeQTable* el que escribirá la misma en el fichero *txt*. La función *update* calcula el nuevo *valor Q* utilizando la función no determinista (que es el caso más general), que se basa en el *valor Q* previo y en el estado siguiente. Destacar que estos estados no son los correspondientes aparentemente al *estado Q* que se explicará justo a continuación, sino el estado general del juego, pues esta conversión se hará a la hora de llamar a la función *computePosition* tras una serie de llamadas encadenadas. Por lo que finalmente esa actualización se hará con los *estados q* correspondiente. Además, recibe un parámetro llamado *finishTraining*, que tan solo es un valor *booleano* que indica si el proceso de entrenamiento ha terminado y, por lo tanto, no hay que actualizar la *tabla Q* más veces.

A continuación, tenemos la función clave, que es *QlearningState*. Esta calcula el *estado Q* mencionado en la fase anterior, y para ello se basa en el *gameState* general del juego en un turno concreto. Lo primero que hace esta función es calcular la distancia del fantasma más cercano que esté vivo. Para conseguir este objetivo se utiliza la función que hemos implementado *getNearestGhost*, que va recorriendo la las diferentes distancias que hay de fantasmas al agente y comparándolas, quedándose con la distancia mínima y aprovechando la situación para quedarse también el índice de este fantasma más cercano. Es importante el detalle de que no cuente con los fantasmas muertos, pues estos al morir no desaparecen del mapa, siguen en él aunque el Pac-man no pueda alcanzarlos.

Ahora volvemos a *QlearningState*, donde con el índice devuelto por la función anterior, conseguimos su posición exacta gracias a la función *getGhostPositions*. Una vez la tenemos, la comparamos con la del Pac-man por coordenadas y devolvemos un número identificador de esta situación. En caso de que la coordenada Y del fantasma sea mayor que la del agente significa que se encuentra al norte y la función devolverá 0. Si está al sur, es decir, la coordenada Y es menor, devolverá 1. En cuanto a las coordenadas en X se hace de forma análoga, si es mayor estará al este y se devuelve 2. Si es menor, está al oeste y se devuelve 3.

Cabe destacar que estos estados mencionados anteriormente solo se mantienen así si en la posición adyacente del Pac-man hacia el fantasma objetivo no hay una pared. En caso de que haya una, el estado variará. Por ejemplo, si el fantasma está al norte, pero hay una pared y no se puede realizar esa acción, el estado pasará a ser este u oeste dependiendo de si el fantasma objetivo se encuentra en una dirección u otra. Así sería sucesivamente si en la dirección elegida también hay un muro, para acabar moviéndose hacia el sur en caso de que no se pueda en ninguna otra dirección. Esta acción se deja como última opción, pues sería la que más aleja al agente de acercarse al fantasma que está al norte. Con el resto de posibles posiciones relativas se haría de manera análoga.

Para que toda esta lógica y el nuevo estado funcione correctamente se han modificado dos funciones del tutorial 4: *computePosition* y *getAction*. El resto se han mantenido sin cambios.

La primera calcula la fila de la *tabla Q* para un estado dado, en nuestro caso es fácil pues cada fila coincide con el número devuelto al calcular el *estado Q*. Es decir, si el estado es 0, el fantasma más cercano se encuentra al norte, la fila será cero, si es uno la fila será uno y así sucesivamente.

Por otro lado, en cuanto a la función *getAction* se han realizado los cambios que se detallarán a continuación. Esta calcula la acción a tomar en un determinado estado. Se divide en dos bloques principales, el que la acción elegida sea aleatoria y en la que se elige según la política previamente aprendida. La decisión de entrar en un bloque de código u otro se elige con una probabilidad *epsilon.* Exactamente, esta será la probabilidad de ejecutar una acción aleatoria, lo cual ayuda a explorar nuevos estados y descubrir, en caso de que existan, mejores políticas. Se elija lo que se elija, el código que se ejecutará es similar. Este se basa en comprobar si la acción realizada es igual al *estado Q* calculado, es decir si el estado es 0, el fantasma se encuentra al norte, y la acción realizada es moverse al norte se dará un refuerzo de 1. Posteriormente a evaluar esto se llama a la función *update* previamente explicada. Como argumentos de este método, a destacar encontramos el estado siguiente y el actual, para el cual utilizamos la

variable *lastState* que se actualiza con el nuevo estado después de realizar la llamada a la función *update*. Con esto se consigue manejar dos estados, el anterior y el de este turno.

Finalmente, al terminar el episodio, se llamará al método *writeQTable*, que escribirá los valores actualizados de la tabla "en memoria" al fichero correspondiente. De esta manera, al arrancar nuevamente el agente, se partirá de lo aprendido anteriormente ya escrito y actualizado.

EVOLUCIÓN HISTÓRICA

Antes de llegar a implementar el agente mencionado anteriormente, valoramos y trabajamos con otros diferentes que se van a describir a continuación. Se siguió un proceso de desarrollo incremental como se aconsejaba en el enunciado. Este consistía en ir probando en los 5 mapas proporcionados (*labAA1.lay* … *labAA5.lay*) de manera que sólo se ascendía al siguiente nivel cuando ya se solucionaba el anterior a la perfección.

En primer lugar, utilizamos uno basado en una función de refuerzo que utilizaba la puntuación actual y la anterior y las restaba para utilizarlo como refuerzo. Esta opción nos pareció la más lógica, pues de por si el Pac-man ya es un juego que recompensa realizar ciertas acciones con un peso determinado, como comer un fantasma o un punto de comida. Además, obtener la máxima puntuación es el objetivo principal del juego. Aun con todo esto a favor, a la hora de implementarlo no conseguimos los resultados esperados. El agente no terminaba de aprender bien y si lo hacía aprendía muy lentamente. Debido a esto decidimos cambiar de estrategia.

El siguiente agente que codificamos era uno más similar al final, este se basaba en un estado con la posición relativa del fantasma más cercano. Los resultados a la hora de testearlo fueron buenos, el agente aprendía bien y sorprendentemente rápido. El inconveniente vino cuando lo probamos en un mapa con paredes. En caso de que hubiera una pared entre el camino óptimo aprendido y el fantasma el agente no reaccionaba ante ese inconveniente y entraba en bucles infinitos sin conseguir alcanzar su objetivo. Fue aquí cuando nos dimos cuenta de que teníamos que sortear las paredes de alguna manera. Entonces utilizamos la estrategia explicada anteriormente en la descripción del agente final, cambiar el estado dependiendo de si en la posición adyacente hacia el fantasma objetivo estaba ocupada por una pared. Esto mejoraba mucho el comportamiento del agente por lo que optamos por esa opción.

Finalmente, para mejorarlo aún más y para que este pueda salir de las situaciones más comprometidas como esquinas de muros, se implementó la lógica expuesta previamente. En esta se evalúa en todos los casos posibles si una acción hacia la posición del fantasma más

cercano es legal, es decir no existe un muro. Esto se haría sucesivamente cambiando de estado de mejor opción hasta la peor en caso de ser la única salida posible para el agente, por mucho que le aleje de su objetivo. Tras todos estos cambios conseguimos el agente final, el cual aprendía correctamente, teniendo en cuenta los posibles obstáculos del camino y saliendo de situaciones comprometidas.

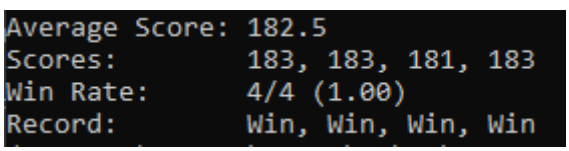
4.- FASE 3. EVALUACIÓN DEL AGENTE

Finalmente, se realizará un análisis de los resultados obtenidos por el agente final implementado con distintas configuraciones y en distintos mapas diferentes, justificando por qué ha funcionado bien el agente seleccionado. Además, se describirán posibles mejoras que se pueden realizar para aumentar el rendimiento.

En primer lugar, se va a comparar el agente actual con el de la práctica 1, el cual funcionaba con modelos de predicción y clasificación. Esto se hará progresivamente, empezando con mapas de dificultad fácil sin paredes y con pocos fantasmas para ir aumentando la dificultad a medida que se añaden más objetivos y se colocan obstáculos antes de llegar a ellos. Después de cada análisis se mostrarán capturas de pantalla de los resultados obtenidos en cada tipo de mapa.

ANÁLISIS Y EVALUACIÓN

Al probar con los mapas más básicos ya pudimos apreciar que nuestro nuevo agente superaba con creces al de la práctica 1. Este era mucho más inteligente y completaba los mapas con rapidez, a la vez que aprendía sorprendentemente rápido. Este rápido aprendizaje y buen funcionamiento es debido a que la función de refuerzo para el estado definido es muy adecuada. En estos casos no es necesario tener en cuenta posibles paredes por lo que el agente es recompensado exactamente como planeamos, es decir, dando más valor a las acciones que le acerquen hacia el fantasma más cercano. Todo esto no pasaba con el agente de la práctica 1 porque no hicimos un profundo tratamiento y filtrado de las instancias y atributos por lo que no conseguimos un modelo bueno para este juego, quizá por nuestra falta de experiencia en este ámbito.



```
Average Score: 381.0
Scores:        383, 379, 383, 379
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

```
Average Score: 622.0
Scores:        709, 417, 713, 649
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

```
Average Score: 382.5
Scores:        390, 382, 378, 380
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

A continuación, pasamos a evaluar el Pac-man en mapas de dificultad media, aquí el número de fantasmas aumentaba y se incluían algunas paredes. Pudimos ver que el agente conseguía generalizar en estos mapas correctamente, aunque le costaba un adaptarse al nuevo entorno, aun así completaba varios de los escenarios. Esto es debido a que nuestra función de refuerzo se centra en reforzar acciones de una manera muy directa, quizá teniendo en cuenta la puntuación habríamos conseguido generalizar más y centrarnos verdaderamente en los movimientos más importantes para maximizar la misma. En cuanto a nuestro agente de la práctica 1 en estos mapas era mucho peor, pues no logramos un modelo entrenado correctamente para evitar paredes y elegir sus movimientos en base a varios objetivos.

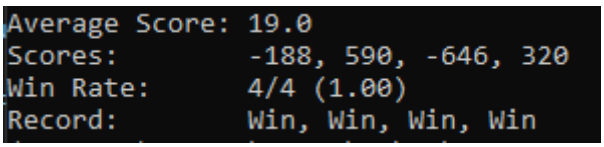
```
Average Score: 3293.0
Scores:        2713, 3425, 3983, 3051
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

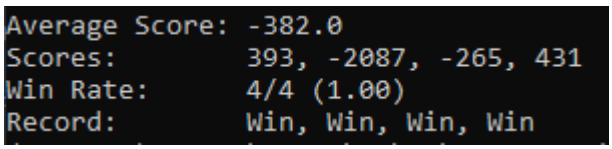
```
Average Score: 170.0
Scores:        177, 25, 273, 205
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

```
Average Score: 1756.25
Scores:        2577, 1569, 2128, 751
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

```
Average Score: 231.5
Scores:        281, 143, 245, 257
Win Rate:      4/4 (1.00)
Record:        Win, Win, Win, Win
```

En último lugar, pasamos a probar nuestra implementación en los mapas más complejos llenos de paredes y con múltiples fantasmas. En este caso nuestro agente no conseguía realizar un aprendizaje óptimo, pues consideramos que nuestra lógica para esquivar muros no compactaba a la perfección con la función de refuerzo seleccionada. Por lo que nuestra decisión fue aumentar el valor de epsilon para permitir al agente explorar más en busca de nuevas rutas. De esta manera conseguimos completar algunos mapas, aunque los resultados no eran los mejores. Sin embargo, como viene pasando en el resto de las pruebas que hemos hecho, el agente tenía un comportamiento notablemente mayor al del basado en aprendizaje supervisado.





POSIBLES MEJORAS FUTURAS

Para que los resultados analizados sean superiores, como posibles mejoras de cara al futuro se podría hacer una mezcla de los agentes por los que hemos ido pasando hasta llegar al final. Esto lo que haría sería tener como función de refuerzo, la definitiva que elegimos sobre recompensar acciones que sean iguales a la posición relativa del fantasma más cercano al agente y a la vez tener en cuenta la puntuación. Esta puntuación ponderaría ese refuerzo dando más o menos recompensa si en ese turno en concreto se ha mejorado la misma. Creemos a su vez que, si elegimos un conjunto de atributos más completo para definir un estado, la función de refuerzo basada en la puntuación que intentamos implementar al principio de la práctica habría funcionado mejor, pues al fin y al cabo en este juego lo importante es conseguir la puntuación más alta posible. Para ello, por ejemplo, se podrían tener en cuenta los puntos de comida, los cuales hemos ignorado en el desarrollo de nuestro agente. Por último, si quisiéramos mejorar aún más el rendimiento consideramos que nuestra lógica para evitar paredes todavía tiene un margen de mejora.

5.- CONCLUSIONES

Para concluir, se añadirán unas apreciaciones generales sobre las prácticas de la asignatura en general, así como unas conclusiones sobre las tareas realizadas y una descripción de los problemas encontrados. Además, se añadirán unos comentarios personales y una opinión acerca de la práctica y las dificultades que se han observado.

APRECIACIONES GENERALES

En primer lugar, comentaremos las apreciaciones generales como para qué pueden ser útiles los modelos obtenidos u otros dominios posibles en los que aplicar aprendizaje por refuerzo. En nuestra opinión, consideramos que tanto el modelo de aprendizaje por refuerzo como el modelo de clasificación y de regresión implementados en la práctica 1 son muy útiles en muchos aspectos de la vida, y es que la inteligencia artificial está cobrando cada vez más importancia en el mundo actual en el que vivimos. Un ejemplo muy sencillo es la aspiradora robot, que ya se ha abierto paso en muchos hogares y que cuenta con un sistema automático que le permite realizar su función y superar distintos obstáculos a su paso.

Además del ámbito de los videojuegos y del ejemplo previamente mencionado, otros posibles dominios en los que creemos que el aprendizaje por refuerzo podría ser muy útil es el de sistemas de navegación o el de gestión de recursos. En el primero ya se han desarrollado coches autónomos, como el de Google, y robots que utilizan esta tecnología para manejarse sin necesidad de una persona real. En cuanto a la gestión de recursos, de nuevo tenemos a Google como ejemplo, que usa una herramienta llamada *DeepMind* para reducir el consumo energético de sus *Data Centers*. De hecho, podríamos afirmar incluso que casi cualquier problema puede ser solucionado mediante aprendizaje por refuerzo si se consigue formalizar correctamente como un MDP (*Markov Decision Process*).

CONCLUSIONES Y PROBLEMAS ENCONTRADOS

Centrándonos ahora en esta práctica concretamente, creemos que la tarea a realizar ha sido muy útil para fijar los conceptos teóricos expuestos en clase. Nos sirve para comparar los distintos tipos de aprendizaje más importantes al contraponerse con la práctica 1, en la que se usaba aprendizaje supervisado mediante clasificación y regresión.

En cuanto a los problemas encontrados en la práctica, destacamos el comienzo de la misma. Al ser un código extenso y no implementado por nosotros en su completitud, nos encontramos

con varios problemas en el mismo a la hora de hacer la fusión entre el del tutorial 1 y el tutorial 4. Creemos que esto se puede mejorar de cara a años posteriores, pues el tiempo que se dedica es mucho a pesar de que no es el objetivo principal de la práctica.

Una vez que se consigue tener un código ejecutable, el desarrollo se agiliza mucho, aunque también es cierto que saber que función modificar para conseguir tu objetivo se hace un poco tedioso. Creemos que en la práctica 1 el enunciado estaba más claro en ese aspecto. Otro pequeño problema que encontramos era el de no poder ejecutar varios episodios seguidos para entrenar al Pac-man; se podía, pero la *tabla Q* no se actualizaba correctamente. Comunicamos esto a nuestro profesor y nos solucionó el problema.

Cuando se resolvió todo esto, el resto de problemas fueron más enfocados a implementar nuestras ideas en cuanto a estados y función de refuerzo elegida. Al tratarse de la parte del tutorial 4, no estábamos tan familiarizados con el código y no comprendíamos bien dónde y cómo hacer cada cosa. Pero en este punto el avance fue mucho más rápido que en fases anteriores.

En definitiva, una práctica útil para conseguir entender y fijar los contenidos vistos en las clases teóricas, con una estructura progresiva que sirve para ello. En contraposición, creemos que alguna indicación más en cuanto a cómo conseguir un código inicial ejecutable, ayudaría mucho a los alumnos a la hora de empezar con la tarea.