Entrenando bots para juegos mediante aprendizaje por refuerzo basado en casos

Fernando Domínguez Estévez

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO DEL DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Director: Antonio Sánchez Ruiz-Granados
Codirector: Pedro Pablo Gómez Martín

Junio 2017

Abstract

Los videojuegos son un interesante campo de estudio para muchos investigadores de inteligencia artificial, dado que multitud de algoritmos distintos pueden ser estudiados y probados con ellos, y luego estos pueden ser aplicados a muchas otras situaciones. En este proyecto se utilizan principios de razonamiento basado en casos y aprendizaje por refuerzo para entrenar bots y que jueguen a Ms.Pac-Man. En concreto se utiliza el algoritmo Q-learning reemplazando la tabla Q y los estados por una base de casos. El uso de casos permite lidiar con una completa representación del estado del juego, y utilizar conocimiento experto sobre el dominio del juego tanto en la recuperación de casos como en la adaptación de soluciones.

Palabras clave: Aprendizaje por Refuerzo, Razonamiento Basado en Casos, Videojuegos, Bots, PacMan, Inteligencia Artificial.

Video games are an interesting field of study for many artificial intelligence researchers, since many different AI methods can be studied and tested with them, and later those investigations can be applied to many other situations. In this project case based reasoning and reinforcement learning principles are used to train bots to play the Ms. Pac-Man vs. Ghosts game. In particular, we use the well-known Q-learning algorithm but replacing the Q-table and states with a case base. The use of cases allows us to deal with rich game state representation and inject domain knowledge in both the retrieval and the adaptation stages.

Keywords: Reinforcement Learnign, Case Based Reasoning, Video Games, Bots, Pac-Man, Artificial Intelligence.

Índice general

Índice de figuras IX

Índice de cuadros XI

1. Introducción 1

1.1. Contexto . . . 1
1.2. Objetivos . . . 2
1.3. Estructura del documento . . . 2

2. Aprendizaje por refuerzo y razonamiento basado en casos 5

2.1. Aprendizaje por refuerzo . . . 5
2.1.1. Meta y recompensas . . . 6
2.1.2. Estados y la propiedad de Markov . . . 7
2.1.3. Procesos de decisión de Markov . . . 7
2.1.4. Funciones de valoración . . . 7
2.1.5. Políticas óptimas . . . 9
2.1.6. Aprendizaje por diferencias temporales . . . 10
2.1.7. Ventajas de TD frente a Monte Carlo y programación dinámica . . . 10
2.1.8. Q-learning . . . 11
2.1.9. La tabla Q . . . 11
2.1.10. Equilibrio exploración-explotación . . . 12
2.1.11. Ventajas e inconvenientes . . . 12
2.2. Razonamiento basado en casos . . . 13
2.2.1. Casos . . . 14
2.2.2. Recuperación de casos . . . 15
2.2.3. Solución a un caso . . . 16
2.2.4. Adaptación . . . 16
2.2.5. Evaluación . . . 16

2.2.6. Gestión de la base de casos 16
2.2.7. Ventajas y desventajas del CBR 17

3. IA en videojuegos **19**
3.1. Entornos de IA en videojuegos 19
3.2. Descripción del juego 22
3.3. Framework utilizado 24

4. Implementando bots para Ms. Pac-Man **27**
4.1. Estructura del bot 27
4.2. Modelando el estado de juego 28
4.3. Recompensas 32
4.4. Espacio de acciones 33
4.5. Q-learning y Pac-Man 34
4.5.1. Memoria de estados 34
4.5.2. La tabla Q 34
4.5.3. Agente 35
4.5.4. Algoritmo Q-Learning 35
4.6. Mezclando RL y CBR 36
4.6.1. El Caso y la Base de Casos 36
4.6.2. Medidas de distancia y similitud 39
4.6.3. Recuperación de casos y nuevos casos 42
4.6.4. Adaptación de soluciones 42
4.7. Gestión de la Base de Casos 43
4.8. Mejorando el rendimiento 44

5. Experimentos y resultados **47**
5.1. Experimentos y resultados de RL 47
5.1.1. Experimento 1 48
5.1.2. Experimento 2 49
5.2. Experimentos y resultados de RL/CBR 50
5.2.1. Experimento 1 51
5.2.2. Experimento 2 51
5.2.3. Experimento 3 52

6. Conclusiones y trabajo futuro **57**
6.1. Conclusiones 57

6.2. Trabajo futuro 58
6.3. Difusión 59

Bibliografía **61**

Índice de figuras

2.1. Ciclo CBR. 15

3.1. OpenAI Universe. 20
3.2. Pantalla del juego Starcraft. 21
3.4. Personaje del jugador en Ms. Pac-Man. 22
3.3. Pantalla del juego Ms. Pac-Man. 23
3.5. Fantasmas en Ms. Pac-Man. 23

4.1. Píldoras cercanas a Pac-Man. 29
4.2. Fantasmas cercanos a Pac-Man. 30
4.3. Píldoras especiales cercanas a Pac-Man. 31
4.4. Fantasmas comestibles cercanos a Pac-Man. 32
4.5. Intersecciones cercanas a Pac-Man. 33
4.6. Ciclo Q-learning 36
4.7. Ejemplo de caso 38

5.1. Puntuaciones del experimento 1. 48
5.2. Tiempos del experimento 1. 49
5.3. Puntuaciones del experimento 2. 50
5.4. Tiempos del experimento 2. 50
5.5. Puntuaciones del experimento 1. 52
5.6. Tiempos del experimento 1. 53
5.7. Puntuaciones del experimento 2. 54
5.8. Tiempos del experimento 2. 55
5.9. Puntuaciones del experimento 3. 55
5.10. Tiempos del experimento 3. 56

Índice de cuadros

2.1. Ejemplo de tabla Q 12

4.1. Ejemplo de tabla Q en Pac-Man 34
4.2. Ejemplo de similitud de casos 41

5.1. Estadísticas de los experimentos con CBR 54

Capítulo 1

Introducción

1.1. Contexto

Actualmente la Inteligencia Artificial está ganando mucho protagonismo en nuestras vidas. Continuamente vemos noticias sobre nuevas aplicaciones de esta rama de la computación y de cómo estas están transformando la sociedad, desde motores de búsqueda, robots que se encargan de la recepción de un hotel, coches que se conducen solos, o apps para teléfonos móviles que permiten descubrir qué canción está sonando o conocer el estado del tráfico en cada momento.

La tecnología está cambiando nuestra forma de ver el mundo a un ritmo frenético gracias a especialistas cuyo trabajo es desarrollar nuevos programas y algoritmos que permitan que eso ocurra, y hacerlo además de forma que esa tecnología sea segura para quien la utiliza.

Muchos de esos algoritmos están relacionados con la forma en que las máquinas pueden aprender y razonar. Uno de los grandes objetivos de la Inteligencia Artificial es estudiar si los ordenadores pueden llegar a pensar como los humanos en ciertos aspectos, tarea que los investigadores llevan persiguiendo durante años, y que últimamente está consiguiendo logros que hasta ahora se consideraban propios de la ciencia ficción.

Los videojuegos representan en muchos casos un entorno óptimo para poder estudiar y probar este tipo de algoritmos, pues conforman un sistema que puede dar pie a multitud de situaciones y problemas distintos, según qué videojuego se esté estudiando, y por tanto pueden surgir muchas soluciones distintas como respuesta, que a su vez se podrán aplicar a muchos otros campos. Por ejemplo, desarrollar una IA para un videojuego de conducción como Gran Turismo puede luego expandirse hasta una IA que sea capaz de conducir en el mundo real.

1.2. Objetivos

El principal objetivo de este proyecto es investigar las aplicaciones del aprendizaje por refuerzo y el razonamiento basado en casos en el ámbito de los videojuegos. Esto, por supuesto, puede tener luego muchas y diversas aplicaciones en el mundo real, dado que los videojuegos a veces consisten en simulaciones con bastante detalle de situaciones reales.

En este caso el elegido ha sido PacMan, conocido por todos al ser uno de los videojuegos más famosos e importantes de la historia. Para alcanzar el objetivo de dominar este juego en concreto, se fueron fijando subtareas más pequeñas, para ir cumpliéndolas y ver así la progresión del bot.

Se podría decir muy brevemente que el proceso seguido ha sido el siguiente:

- **Estudiar técnicas de aprendizaje por refuerzo (RL)**, que incluyen algoritmos como Sarsa y Q-learning, siendo este último el que se utilizó finalmente. También se empezó a practicar con estos algoritmos en problemas más simples que PacMan, para asegurarnos que se entendía e implementaba correctamente.

- **Implementar aprendizaje por refuerzo en PacMan**, y estudiar los problemas y sobre todo las limitaciones de usar estas técnicas en un juego que resultó ser más complejo de lo que en un principio parecía.

- **Estudiar técnicas de razonamiento basado en casos (CBR)**, para poder mezclarlas con lo que ya se tenía de aprendizaje por refuerzo y mejorar el rendimiento del bot.

- **Implementar CBR con PacMan**, de forma que trabajen CBR y RL de manera conjunta. Este paso conforma la parte central del proyecto, pues suponen desarrollar un framework de aprendizaje, y estando solo una persona para la implementación, con lo que sería una difícil tarea.

- **Realizar experimentos y analizar resultados**. Este paso era crucial para saber cómo de bueno es el rendimiento del bot, y si juega de manera inteligente o no. Tanto si la respuesta era afirmativa como negativa, analizar los resultados y buscar explicaciones es la base para seguir desarrollando este trabajo en el futuro.

1.3. Estructura del documento

Esta memoria contiene toda la información necesaria para entender el software que se ha desarrollado, desde lo más básico.

Está dividido en capítulos, que son descritos brevemente a continuación:

- **Capítulo 2 - Aprendizaje por refuerzo y razonamiento basado en casos:** En este capítulo se encuentra toda la teoría necesaria para entender el proyecto, es decir, los fundamentos básicos de los algoritmos, necesarios para comprender lo que se cuenta en capítulos posteriores.
- **Capítulo 3 - IA en videojuegos:** Aquí se comentan los diversos entornos destinados al desarrollo de inteligencias artificiales en videojuegos.
- **Capítulo 4 - Implementando bots para Ms. Pac-Man:** Este es el capítulo central del proyecto, en el que se desarrolla el proceso seguido para implementar todos los métodos comentados anteriormente, los experimentos realizados y los resultados analizados y comentados.
- **Capítulo 5 - Experimentos y resultados:** En este capítulo se comentan diversos experimentos realizados para poner a prueba las técnicas estudiadas anteriormente, y se analizan los resultados.
- **Capítulo 6 - Conclusiones y trabajo futuro:** El último capítulo recoge las consideraciones finales extraídas de esta experiencia, y algunas sugerencias de los siguientes pasos a la hora de continuar con esta investigación.

El código completo de este proyecto puede encontrarse en: https://github.com/fedomi/PacMan-vs-Ghosts.git.

Capítulo 2

Aprendizaje por refuerzo y razonamiento basado en casos

Introducción

Se pretende por tanto en este proyecto explorar maneras de implementar bots que actúen de forma inteligente, para lo cual extraerán información de su entorno, y la usarán en su propio beneficio. Hay muchas y muy variadas técnicas de inteligencia artificial que permiten esto, pero nos centraremos en dos de ellas: el *Aprendizaje por Refuerzo*, y el *Razonamiento Basado en Casos*.

Estas serán descritas teóricamente y de manera general en este capítulo, y posteriormente veremos su aplicación de manera conjunta para implementar bots inteligentes en el ámbito de los videojuegos.

2.1. Aprendizaje por refuerzo

El primer método abordado fue el de aprendizaje por refuerzo, o 'Reinforcement Learning' (RL). Este método intenta resolver el problema de escoger qué acciones debe realizar el aprendiz (o *agente*) en cada situación a la que se enfrente, para conseguir maximizar una recompensa numérica. A dicho aprendiz no se le dice directamente qué acciones debe realizar, sino que debe descubrirlas por si mismo, probándolas, y así averiguar cuáles le permiten obtener una mayor recompensa. Estas acciones, por supuesto, no solo afectan a la recompensa inmediata, sino que también influirán en las recompensas posteriores.

Estas dos características (prueba-error, y recompensas), son las dos cualidades principales del aprendizaje por refuerzo.

Dicho aprendiz interactuará con el entorno que le rodea en base a la representación que tenga de él, es decir, los datos que conozca acerca de qué le rodea, que le servirán para distinguir unas situaciones de otras. Este entorno irá dando recompensas al agente en función de las acciones que vaya tomando, y dicho agente centrará su atención en elegir las mejores acciones que maximicen estas recompensas.

Más concretamente, el agente interacciona con el entorno en una secuencia discreta de tiempos t=0,1,2,3,...; en cada uno de los cuales recibirá una representación del entorno, que llamaremos *estado*, y en función de la cual el agente tendrá que elegir qué acción tomar en cada momento. En el siguiente instante de tiempo, el agente recibirá una recompensa por su acción previa. El objetivo del agente es maximizar la recompensa total obtenida al final.

Esta técnica es muy flexible, ya que puede ser aplicada a multitud de problemas de muchas maneras, si se consigue una representación adecuada del problema. Por ejemplo, los instantes en los que se toman acciones no tienen por qué ser intervalos fijos, sino que pueden referirse a sucesos concretos que le ocurran al agente, en los que tendrá que decidir. Las acciones y estados pueden estar modelados de multitud de formas, con lo que las posibilidades de aplicar este método a problemas muy distintos es enorme.

Por supuesto, las acciones que el agente pueda tomar y la representación del estado influyen enormemente en su proceso de aprendizaje, y por tanto en su rendimiento final, por lo que es conveniente a la hora de implementar los algoritmos que se dedique el tiempo suficiente a modelar la representación del problema y las acciones que se utilizarán para aprender a resolverlo.

En el marco de los videojuegos, que es el que discutiremos en este proyecto, el agente será el personaje que controlaría el jugador, que en este caso intentará aprender a jugar en base a los conocimientos que vaya adquiriendo. El entorno, por tanto, es todo lo que rodea al agente dentro del videojuego, como los enemigos, los ítems que pueda recoger, etc. Las recompensas por tanto será algún tipo de respuesta que el juego le da al agente para que sepa si lo está haciendo bien o no, y así pueda maximizarlo, como pueden ser, por ejemplo, los puntos que acumule durante la partida. Tanto la representación del entorno como las recompensas, por tanto, dependerán de cada videojuego concreto.

2.1.1. Meta y recompensas

En el aprendizaje por refuerzo, la meta del agente está formalizada por medio de las recompensas que le envía el entorno cuando realiza una acción. Esta recompensa es un número que varía en cada instante de tiempo. Normalmente el objetivo del agente es maximizar la cantidad total de recompensa recibida, lo cual implica no sólo fijarse en el premio inmediato, sino en lo que se pueda recibir a la larga.

2.1.2. Estados y la propiedad de Markov

El agente basará sus decisiones en la representación del entorno disponible, que llamamos *estado*. El estado puede contener información de todo tipo sobre lo que rodea al agente dentro del juego, como por ejemplo distancia a los enemigos, posición dentro del mapa, etc. Pero, por otro lado, no es de esperar que el estado informe al agente de absolutamente todo lo que pasa en el juego. El objetivo es que el estado resuma la información relevante que pueda serle de utilidad al agente, y nada más, de forma que ésta sea fácilmente manejable.

Otra característica que deben tener los estados es la llamada *propiedad de Markov*, que se refiere a la propiedad de un estado por la cual las recompensas que se puedan obtener en un futuro a partir de él tomando distintas acciones sucesivas sólo dependen de ese estado, y no de los anteriores. Por ejemplo, en un determinado momento de una partida de ajedrez, las recompensas futuras solo dependen del estado actual del tablero, no de cómo se ha llegado hasta él.

2.1.3. Procesos de decisión de Markov

Una tarea de aprendizaje por refuerzo que satisface la propiedad de Markov para todos los estados se conoce como *proceso de decisión de Markov*, o *MDP*. Si el número de acciones y estados son finitos, entonces se llaman *procesos de decisión de Markov finitos*, o *FMDP*. Estos son particularmente importantes en la teoría del aprendizaje por refuerzo.

Un FMDP particular queda definido por sus conjuntos de estados y acciones. Dados un estado s y una acción a, la probabilidad de cada siguiente estado s' es

$$Pr^a_{s,s'} = Pr\{s_{t+1} = s' | s_t, = s, a_t = a|, \}$$

De igual manera, dado un estado s y una acción a, el valor esperado de la siguiente recompensa será:

$$R^a_{s,s'} = E\{r_{t+1} | s_t, = s, a_t = a, s_{t+1}, = s'|, \}$$

Estas dos cantidades especifican la cualidad más importante de los FMDP.

2.1.4. Funciones de valoración

Casi todos los algoritmos de aprendizaje por refuerzo están basados en estimar funciones de valoración, las cuales valoran cómo de bueno es para el agente estar en cierto estado, o cómo de buena es una acción a realizar, según en qué estado se encuentre. Aquí la noción de

'bueno' depende de las futuras recompensas que se esperan recibir. Por lo tanto, las funciones de valor están definidas en función de las políticas particulares que siga el agente en cuestión.

Una *política* π es una asociación desde un conjunto de estados y acciones a la probabilidad $\pi(s,a)$ de tomar la acción a cuando se está en el estado s. Es decir, establece cómo de probable es tomar cada acción desde un estado. El valor de un estado s bajo una política π, denotado $V^{\pi}(s)$, es la recompensa que se espera obtener cuando se sigue la política π de ahí en adelante. Más formalmente:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\}$$

De manera similar se puede definir el valor de tomar una acción a cuando se está en el estado s siguiendo la política π, denotado $Q^{\pi}(s,a)$, como la recompensa esperada comenzando en s, tomando la acción a, y siguiendo la política π a partir de entonces. Queda definido formalmente como:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\}$$

El valor de las funciones Q y V se puede estimar a través de la experiencia. Por ejemplo, si el agente sigue una política y para cada estado guarda una media de las recompensas que se han obtenido a partir de él, esos valores terminarían convergiendo a los de la función V. Si estas medias se separan para cada una de las acciones posibles que se pueden tomar en cada estado, entonces tendríamos un valor asociado a cada acción a tomar desde cada estado, y estos valores se irían aproximando a los de la función Q, a medida que el agente evoluciona mediante experiencia.

Una propiedad fundamental de las funciones de valoración utilizadas en el aprendizaje por refuerzo y programación dinámica es que satisfacen relaciones recursivas muy particulares. En concreto la ecuación de Bellman, que se detalla a continuación:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\}$$

$$= E_{\pi}\{\textstyle\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\}$$

$$= E_{\pi}\{r_{t+1} + \gamma\textstyle\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s\}$$

$$= \textstyle\sum_a \pi(s,a)\sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma E_{\pi}\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s']$$

$$= \textstyle\sum_a \pi(s,a)\sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^{\pi}(s')]$$

donde s_i y r_i son el estado en que se encuentra el agente en el instante i, $P_{ss'}^a$ y $R_{ss'}^a$ son las probabilidades y recompensas explicadas en el apartado anterior, y γ es un factor de descuento que determina la importancia de las recompensas según lo lejos que estén del momento

actual. Esta ecuación expresa la relación entre el valor de un estado y el valor de los estados que le suceden, y será importante para explicar la idea del siguiente apartado.

2.1.5. Políticas óptimas

Si se ha mencionado la ecuación de Bellman, realmente es para hablar de optimalidad. Y es que resolver un problema mediante aprendizaje por refuerzo significa realmente encontrar una política que de la mayor recompensa posible, es decir, encontrar la *política óptima*.

Denotamos la política óptima por π^*, y de la misma manera las funciones de valoración óptimas serán $V^*(s) = max_\pi V^\pi(s)$ y $Q^*(s,a) = max_\pi Q^\pi(s,a)$. Dado que $V^\pi(s)$ es la función de valoración de cierta política, debe satisfacer la ecuación de Bellman mencionada anteriormente. Como además es la función óptima, puede ser formulada de manera especial sin referirse a ninguna política en concreto:

$$
\begin{aligned}
V^*(s) &= max_a Q^{\pi^*}(s,a) \\
&= max_a E_{\pi^*}\{R_t | s_t = s, a_t = a\} \\
&= max_a E_{\pi^*}\{\textstyle\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \\
&= max_a E_{\pi^*}\{r_{t+1} + \gamma \textstyle\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\} \\
&= max_a \textstyle\sum_s' P_{ss'}^a [R_{ss'}^a + \gamma V^*(s')]
\end{aligned}
$$

Esta ecuación se conoce como la ecuación de optimalidad de Bellman. A pesar de lo complicada que pueda parecer la fórmula, esta ecuación expresa un hecho muy simple, y es que el valor de un estado depende de la mejor acción que se pueda tomar desde él, lo cual tiene mucho sentido. Para la función Q (que recordemos expresa el valor de tomar una acción desde un estado) se tiene:

$$Q^*(s) = \textstyle\sum_s' P_{ss'}^a [R_{ss'}^a + \gamma max_a Q^{\pi^*}(s,a)]$$

Una vez que se tiene V*, resulta relativamente fácil determinar una política óptima. Para cada estado, habrá una o más acciones para las que se obtenga el máximo en la ecuación de optimalidad de Bellman. Por tanto las políticas que elijan aquellas acciones se convierten en políticas óptimas.

2.1.6. Aprendizaje por diferencias temporales

Si hubiera que identificar una idea como principal en el aprendizaje por refuerzo, esta debería ser lo que se conoce como *aprendizaje por diferencias temporales* (*temporal difference learning*, o *TD-learning*). Se trata de una combinación de ideas tanto de los métodos de Monte Carlo como de Programación Dinámica, los cuales se comentan brevemente a continuación:

- **Métodos de Monte Carlo**: Son formas de descubir políticas óptimas para resolver problemas, y están basados en obtener información a partir de la experiencia.
- **Programación Dinámica**: Se trata de una colección de algoritmos que pueden ser utilizados para computar políticas óptimas, dado un modelo del entorno y representando el problema como un proceso de decisión de Markov.

Como los métodos de Monte Carlo, TD puede aprender directamente de la experiencia sin un modelo del entorno y su dinámica. Y al igual que en programación dinámica, TD actualiza estimaciones basadas en parte de otras estimaciones aprendidas, sin esperar al resultado.

Tanto TD como Monte Carlo usan la experiencia para actualizar los valores de las funciones V y Q. Un ejemplo muy simple de Monte Carlo es, en cada episodio, modificar V de la siguiente manera:

$$V(s_t) = V(s_t) + \alpha[R_t - V(s_t)]$$

Mientras que los métodos de Monte Carlo deben esperar al final del episodio para modificar V, TD sólo necesita esperar al final del paso actual. El método TD más simple, conocido como TD(0) es:

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma * V(s_{t+1}) - V(s_t)]$$

2.1.7. Ventajas de TD frente a Monte Carlo y programación dinámica

Obviamente, TD tiene una clara ventaja sobre programación dinámica, y es que no necesitan un modelo del entorno, de sus recompensas y la probabilidad de los estados siguientes. La más evidente ventaja sobre los métodos de Monte Carlo es la mencionada anteriormente, que no necesitan esperar a terminar el episodio completo, sino que se implementan de forma más incremental, y los valores se van actualizando en cada paso dentro de un mismo episodio. Algunas aplicaciones tienen episodios muy largos, por lo que retrasar el aprendizaje hasta el final de los mismos tal vez no sea la mejor opción.

2.1.8. Q-learning

Uno de los más importantes avances en el aprendizaje por refuerzo fue el desarrollo del algoritmo Q-learning. En su forma más simple, se actualiza el valor de la función Q en un paso, de la siguiente manera:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha[r_{t+1} + \gamma * max_a Q(s_{t+1},a_t) - Q(s_t,a_t)]$$

En este caso, la función Q aprendida se aproxima directamente a Q*, la función óptima, independientemente de la política seguida. Esto simplifica enormemente el análisis del algoritmo y las pruebas de convergencia sobre él. El único requisito que debe cumplir la política para garantizar la convergencia del algoritmo es que todos los pares estado-acción deben seguir siendo actualizados de manera continua. El algoritmo Q-learning tiene la siguiente forma:

Algoritmo 1: Algoritmo Q-learning

1 Inicializa $Q(s,a)$ arbitrariamente;
2 **repetir**
3   Inicializa s;
4   **repetir**
5   Elige una acción a desde s mediante una política que depende de Q;
6   Toma la acción a. Observa la recompensa r y el siguiente estado s';
7         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * max_{a'} Q(s',a') - Q(s,a)]$;
8         $s \leftarrow s'$;
9   **hasta que** *s sea terminal*;
10 **hasta que** *no haya más episodios*;

2.1.9. La tabla Q

En el algoritmo vemos que se modifican los valores de la función Q pero, ¿cómo se maneja realmente esta información?

Estos datos se organizan en forma de tabla, en la que se irá añadiendo información sobre los nuevos estados y acciones que se vayan explorando. Así, cada fila de la misma corresponde a un estado distinto, y cada columna almacena información sobre el valor de las acciones. En concreto, el elemento (i,j) de la tabla representa el valor de realizar desde el estado s_i la acción a_j.

Implementar y manejar esta tabla correctamente se convertirá en una prioridad de aquí al final del proyecto pues es donde reside todo el conocimiento del agente sobre las decisiones que puede y debe tomar.

A continuación se muestra un ejemplo:

Estado	Valor acción 1	Valor acción 2	Valor acción 3	...
1	v_{11}	v_{12}	v_{13}	...
2	v_{21}	v_{22}	v_{23}	...
3	v_{31}	v_{32}	v_{33}	...
...	...	...	...	...

Cuadro 2.1 Ejemplo de tabla Q

2.1.10. Equilibrio exploración-explotación

También vemos que en el algoritmo se menciona una “política que dependa de Q”, lo cual quiere decir que el agente debe tomar sus acciones aprovechando lo que ha aprendido previamente. Uno podría pensar que la mejor manera de afrontar una situación pasada es escoger la opción que mejor resultado le haya dado hasta el momento, pero esto puede no ser así. Puede ser que, aunque alguna acción de un buen resultado, haya otra aún inexplorada que, a la larga, le haga obtener al agente una recompensa mayor.

Una de las cualidades, por tanto, que debe tener la política elegida, es que mantenga un equilibrio entre la explotación del conocimiento ya aprendido, y la exploración de nuevas soluciones, para comprobar si efectivamente hay mejores acciones disponibles o no.

Un ejemplo de política que cumple esto, que será muy utilizada de aquí en adelante, es la política ε-*greedy*, que consiste en que en cada momento de decidir qué acción realizar, hay una probabilidad ε de que se realice una acción aleatoria, y si no, se escoge la mejor acción disponible.

2.1.11. Ventajas e inconvenientes

Los métodos de aprendizaje por refuerzo, y en concreto Q-learning, que es el que se ha utilizado, tienen sus ventajas, pero también algunas limitaciones que merece la pena comentar:

- Requieren un estudio del dominio del problema, para modelar correctamente los estados y las recompensas. Si el dominio es difícil de representar de forma que el agente pueda entenderlo, puede que sea complicado implementar este tipo de métodos.

- Son eficaces en problemas pequeños. Si no hay demasiados estados, estos se van a revisitar más a menudo, lo que actualizará la tabla Q a menudo, llegando antes a la solución óptima.
- Por otro lado, una de las mayores limitaciones aparece cuando intentamos diseñar la representación del estado de juego que tiene el agente, y con la que decidirá en qué estado se encuentra en cada momento. Como esta elección se basa comparando exactamente el estado actual con los de la memoria de estados, pueden suceder dos cosas. Primero, si la representación del estado es muy general, puede que no contenga toda la información necesaria para dotar al agente de un conocimiento apropiado. De poco sirve que el agente pase muchas veces por un estado concreto, si este no representa correctamente la realidad de la situación en la que se encuentra. Por otro lado, si el estado es muy específico y está lleno de información, cada momento del juego será más distinguible del anterior, y por tanto se van a generar muchos estados que tienen que ser añadidos a la memoria. Esto plantea además otro problema, y es que al haber tantos estados, estos serán recuperados menos frecuentemente por el agente para su aprendizaje, con lo que sus valores de la función Q no serán actualizados tan a menudo y, o bien convergerán más lentamente a sus valores óptimos (además necesitaremos simular más episodios para que esto ocurra), o puede que la convergencia ni siquiera llegue a producirse.

Si se desea estudiar en profundidad las diferentes técnicas de aprendizaje por refuerzo, un gran punto de partida es [10], que se ha utilizado como una de las principales referencias en la elaboración de este proyecto.

2.2. Razonamiento basado en casos

Viendo las limitaciones del aprendizaje por refuerzo mencionadas en el apartado anterior, preocupa especialmente la última, que hace referencia al problema del tamaño de la tabla Q y la reutilización de estados. En un videojuego mínimamente complejo es habitual que haya muchas situaciones distintas, y por tanto es probable que la tabla Q se nos vuelva difícil de manejar tal y como la conocemos hasta ahora.

Para solventar este problema, se quiere intentar combinar el aprendizaje por refuerzo con otra técnica que ayudará a que siempre tengamos un número razonable de estados entre manos, y que así el agente consiga un aprendizaje de calidad.

Esta técnica es el *Razonamiento Basado en Casos* (*Case Based Reasoning*, o *CBR*), que se explicará en esta sección. Con ella, el agente aprende de situaciones pasadas similares a la

actual, y las utiliza para resolver el nuevo problema. Esto implica adaptar viejas soluciones a las nuevas situaciones que se van dando.

Si se observa la forma en la que las personas resuelven problemas, podemos atisbar un razonamiento basado en casos. Por ejemplo, los abogados usan casos pasados y resueltos como precedentes para justificar sus argumentos en los nuevos casos.

CBR sugiere por tanto un modelo de razonamiento que incorpora comprensión, aprendizaje y resolución de problemas. Este modelo se basa y depende de los siguientes puntos:

- Las experiencias pasadas. Volver a casos antiguos es ventajoso al tratar problemas que incluyen situaciones recurrentes.
- La habilidad para comprender nuevas situaciones a partir de esas experiencias anteriores. Tener una comprensión correcta del problema es un requisito necesario en el proceso de aprendizaje.
- La capacidad de adaptación. Dado que los casos antiguos no suelen ser exactamente iguales a los nuevos, suele ser necesario disponer de una forma de adaptar los problemas y, por tanto, también las soluciones a los mismos.
- La habilidad de integrar nuevos casos en la memoria de forma adecuada. El manejo correcto de la memoria del agente es crucial en este tipo de razonamiento.

2.2.1. Casos

¿Qué se considera un caso entonces? Los casos son representaciones del estado del problema en un cierto momento, y una solución propuesta para resolverlo. Esta solución asociada a cada caso es la que se irá modificando a medida que el agente aprende. Si pensamos en los términos que hemos manejado hasta ahora en la sección de aprendizaje por refuerzo, podemos considerar un caso como un estado del juego, junto a la información de la tabla Q correspondiente, que es la que le dice qué acción le conviene realizar, es decir, es la “solución” a ese estado. Los casos por tanto sirven a dos propósitos:

- Proporcionan sugerencias de soluciones a los problemas.
- Proveen al agente de un contexto con el que entender las situaciones a las que se enfrenta. Todo esto sugiere una serie de procesos que se antojan fundamentales para facilitar el razonamiento. Algunos de ellos son la **recuperación de casos**, el **emparejamiento de casos** con los de la memoria, el **almacenamiento de casos** en memoria, y su gestión.

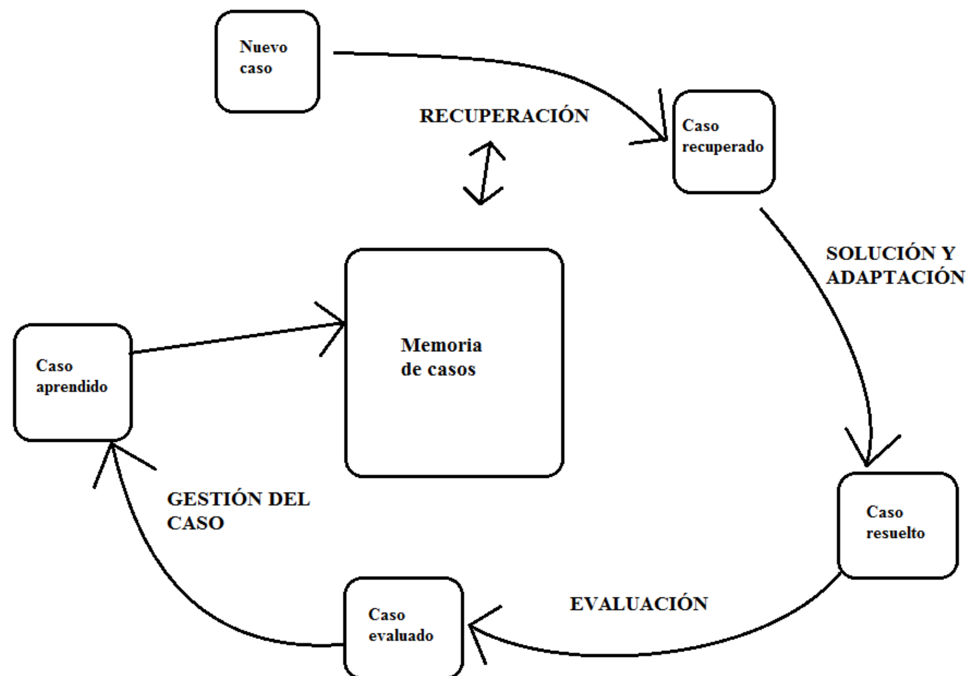

Figura 2.1 Ciclo CBR.

A continuación se explican brevemente los pasos que se dan en un proceso de razonamiento basado en casos.

2.2.2. Recuperación de casos

El objetivo de este paso es recuperar “buenos” casos que puedan aportar información sobre el nuevo caso al que nos enfrentamos. La recuperación se hace comparando el nuevo caso con los de la base de datos según ciertos parámetros que dependen de cómo estén modelados los casos.

Hay algunos grandes problemas que deben comentarse cuando se habla de recuperación de casos. Primero, debemos darle al agente una manera de saber que un caso es aplicable a la nueva situación. Este es el emparejamiento de casos del que se hablaba anteriormente. Cuando parecen iguales el problema no es tan grande, claro. Pero a veces la similitud entre dos casos no es tan evidente, ni la solución de uno tan fácilmente adaptable a la del otro. Las estrategias del ajedrez y videojuegos como Starcraft o Age of Empires, por ejemplo, tienen mucho en común, aunque las características de cada juego en concreto sean bien distintas. Uno se juega en un tablero, el otro a lo largo y ancho de un mapa virtual; uno

tiene piezas, el otro, unidades. Lo que tienen en común es otro tipo de características más abstractas. Hay dos jugadores enfrentados, cada uno quiere ganar y que el otro pierda, y ambos implican planear estrategias en un escenario, y posicionar tus piezas/unidades por los distintos emplazamientos del mismo. Una manera de lidiar con este problema es usar más que la representación superficial de un caso para la comparación.

Otro problema es definir los algoritmos de recuperación. ¿Cómo se puede buscar en una enorme base de casos de forma eficiente para encontrar casos apropiados?

Estos y otros inconvenientes que surgen en la recuperación conforman el **problema de indexado de casos**. Este es el problema de recuperar casos aplicables en un tiempo razonable.

2.2.3. Solución a un caso

En el siguiente paso, las porciones relevantes de los casos recuperados son extraídas para formar una solución al nuevo caso. Normalmente este paso consiste en seleccionar la solución al problema antiguo, o una parte de ella, y utilizarla en el nuevo.

2.2.4. Adaptación

Las soluciones a casos antiguos sirven de inspiración para solucionar casos nuevos, pero rara vez ambos problemas son exactamente iguales, por lo que estas soluciones deben ser modificadas. En este paso, llamado adaptación, la solución antigua es arreglada para encajar con el nuevo problema. Esta tarea compone dos procesos principales: averiguar qué necesita ser modificado, y llevar a cabo la adaptación.

2.2.5. Evaluación

En el siguiente paso, los resultados del razonamiento son puestos a prueba para obtener feedback. Este es estudiado y analizado, para ver si los resultados son los esperados. Si no lo son, un análisis posterior es necesario, para averiguar la causa y ponerle remedio. La evaluación es el proceso de juzgar cómo de buenas son las soluciones propuestas. Este proceso incluye explicar las diferencias entre lo que se esperaba y lo que realmente ocurre, de manera justificada, y proponer y comparar posibles alternativas.

2.2.6. Gestión de la base de casos

El nuevo caso es almacenado en la base de casos para un uso posterior. El proceso más importante llegado a este punto es elegir las formas en que se indexan los nuevos casos. Este

indexado debe hacerse de tal forma que el nuevo caso pueda ser utilizado de nuevo en un razonamiento futuro, cuando sea necesario.

2.2.7. Ventajas y desventajas del CBR

El razonamiento basado en casos tiene muchas ventajas para el agente:

- CBR permite al agente proponer soluciones a problemas rápidamente, evitando el tiempo necesario para construir esas soluciones de la nada.
- También le permite proponer soluciones en dominios que no son completamente entendidos por él.
- Le da al agente una forma de evaluar soluciones cuando no hay un método algorítmico disponible para ello.
- Recordar experiencias previas es particularmente útil para el agente, pues implica que recuerda los problemas que han ocurrido en el pasado, y por tanto le sirve para evitar repetir errores anteriores.
- Los casos ayudan al agente a centrarse en las características importantes del problema, las más determinantes para desarrollar una solución.

Por supuesto, también existen algunas desventajas:

- Un agente que utilice CBR puede estar tentado de usar los casos antiguos ciegamente, confiando en la experiencia previa sin validar la nueva situación.
- La asociación entre nuevas situaciones y casos antiguos puede ser problemática. Si no se realiza correctamente, el conocimiento que se adquiere puede no ser óptimo o incluso ser incorrecto.

Capítulo 3

IA en videojuegos

Introducción

La inteligencia artificial aplicada a videojuegos es un tema cada vez más popular, con lo que a lo largo de los años han ido surgiendo en la comunidad diferentes entornos en los que poder aplicar todo tipo de algoritmos a juegos de multitud de plataformas, desde los clásicos, a los más modernos.

En este capítulo se comentarán algunos de ellos, y en especial el que ha sido usado para este proyecto. También se hablará de cómo es el videojuego escogido, Ms.Pac-Man, y de las peculiaridades que tiene la versión del mismo en el framework utilizado.

3.1. Entornos de IA en videojuegos

Como ya se ha comentado, en los últimos años se han ido creando diferentes frameworks en los que programar IA para videojuegos, debido al creciente interés por estas aplicaciones que se percibe entre los investigadores.

Al principio del desarrollo se tuvieron varias opciones en cuenta a la hora de qué entorno se iba a utilizar para desarrollar el proyecto. Una de las opciones que se dejó fuera, pero que también era muy interesante es alguna de las herramientas proporcionadas por OpenAI, una organización sin ánimo de lucro fundada, entre otros, por Elon Musk, que se dedica a la investigación en el campo de la inteligencia artificial.

Su principal objetivo es impulsar a los desarrolladores a que investiguen en esta área para que en el futuro las inteligencias artificiales sean seguras para el ser humano.

Comenzaron su tarea lanzando *OpenAI Gym*, unas pequeñas herramientas destinadas a probar algoritmos en problemas sencillos, como el popular videojuego Pong, un simulador de aterrizaje lunar, o versiones simplificadas de algunos juegos de mesa.

Estas primeras herramientas se pueden encontrar en: https://gym.openai.com/

Ahora, con su nuevo proyecto, *Universe*, han dado un paso más allá, proporcionando herramientas más avanzadas y siendo posible aplicarlas a videojuegos más complejos, como Portal o Civilization.

Se puede obtener más información sobre Universe en: https://universe.openai.com/

Figura 3.1 OpenAI Universe.

Otro indicador de que es este tipo de investigaciones están en auge se puede ver en las competiciones cada vez más frecuentes que se realizan de inteligencias artificiales en videojuegos, como pueda ser, por ejemplo, las competiciones de AIIDE (*Artificial Intelligence and Interactive Digital Entertainment*) que se realizan entre inteligencias artificiales desarrolladas para el popular videojuego de estrategia StarCraft.

AAIDE es el punto de encuentro de investigadores de IA de todo el mundo, y desarrolladores de software de entretenimiento como pueden ser videojuegos, y consiste en un evento repleto de conferencias, presentaciones y talleres de todo tipo relacionado con estos temas.

Desde su página web se puede descargar la API específica que hay que utilizar en la competición, y empezar a implementar bots en C++ para jugar a StarCraft. Estos bots serán puestos a prueba en la competición, en los que jugarán unos contra otros.

Figura 3.2 Pantalla del juego Starcraft.

El propósito de esta competición es evaluar el progreso de la investigación de inteligencias artificiales aplicada a juegos de estrategia en tiempo real (como es el caso de Starcraft). Estos juegos plantean un desafío mayor para los investigadores que, por ejemplo, el ajedrez, debido a que hay información oculta para cada jugador, a que hay enormes espacios de estados y acciones posibles, y el requisito de que los bots tienen que actuar rápidamente. En este tipo de juegos los mejores jugadores humanos aún conforman el bando ganador, pero esto probablemente cambie en los próximos años, gracias a competiciones como esta.

Se puede encontrar más información acerca de esta competición en: http://www.cs.mun.ca/~dchurchill/starcraftaicomp/index.shtml

Otra competición, esta vez para el juego Pac-Man, es la organizada por IEEE CIG (*Computational Intelligence in Games*), como se puede ver en [1]. En ella, los bots enviados son puestos a prueba un número prefijado de veces, y el que consiga la mayor puntuación máxima en estos intentos, será el ganador.

Otro entorno para desarrollar IA, al cual se puede acceder igualmente desde [1], es el desarrollado por un usuario llamado *kefik*, destinado a la programación para el videojuego Ms.Pac-Man, con la intención de realizar competiciones entre bots, como ya se mencionaba

antes con IEEE CIG. Es el entorno que se utilizará en este proyecto, debido a que Pac-Man es un juego lo suficientemente complejo como para que no sea trivial resolverlo. Tiene muchos matices y se pueden explorar multitud de técnicas con él. Por otro lado, juegos de estrategia en tiempo real como Starcraft se antojan demasiado complicados para el propósito de este trabajo, serían necesarios un equipo más grande, o disponer de más tiempo, para explorar ese tipo de videojuegos en profundidad.

La versión base del entorno que utilizaremos se puede descargar en el siguiente enlace:

https://github.com/kefik/MsPacMan-vs-Ghosts-AI.

Se trata de un simulador muy completo de Ms.Pac-Man, que proporciona todo lo necesario para programar bots en Java. Se puede obtener más información sobre el desarrollo de este software en:

https://www.facebook.com/pacman.vs.ghosts/.

A continuación se explicará en qué consiste este popular juego, y las características especiales de este simulador en concreto.

3.2. Descripción del juego

Se ha escogido Ms. Pac-Man para experimentar con el aprendizaje por refuerzo y el razonamiento basado en casos ya que es un videojuego especialmente interesante para aplicar estas técnicas, debido a su enorme cantidad de variables, y a los problemas y obstáculos que plantea, si bien es cierto que esta magnitud en los problemas ha incrementado enormemente la dificultad de la implementación y la obtención de resultados.

El protagonista de Ms.Pac-Man es el personaje amarillo, que debe ir superando distintos laberintos, en los que su objetivo es sobrevivir sin ser devorado por los fantasmas que deambulan por él, mientras recoge píldoras y comida, que le darán puntos.

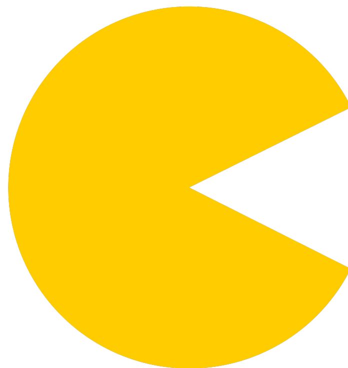

Figura 3.4 Personaje del jugador en Ms. Pac-Man.

Figura 3.3 Pantalla del juego Ms. Pac-Man.

Dichos fantasmas vagan por el laberinto en busca del jugador. Cada vez que los fantasmas comen a Pac-Man, le quitan una vida y le devuelven a la posición de salida, y vuelven todos a la guarida, de la que saldrán pasados unos segundos para volver a buscar al jugador.

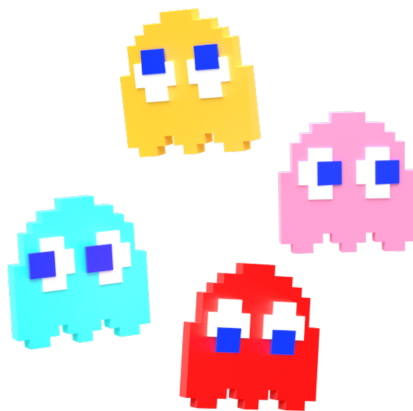

Figura 3.5 Fantasmas en Ms. Pac-Man.

El jugador tiene que ir recogiendo píldoras que se muestran como pequeños círculos blancos en el laberinto, las cuales le darán puntos. Existen varias píldoras más grandes, que

darán al jugador durante unos instantes el poder de comerse a los fantasmas para ganar más puntos. El nivel termina cuando el jugador se come todas las píldoras, o bien cuando pierde todas sus vidas, en cuyo caso termina el juego.

Los puntos por tanto se obtienen en unas pocas situaciones, a saber:

- Cuando se come una píldora, se obtienen 10 puntos
- Cuando se come una píldora especial, se obtienen 20 puntos
- Cuando se come un fantasma, se obtiene una puntuación que se va incrementando si se comen más fantasmas dentro del mismo período desde que se comió la última píldora especial. Inicialmente esta puntuación es de 200 para el primer fantasma, y se dobla cada vez que el jugador se come uno de ellos.
- Cuando el jugador es devorado por un fantasma, no sufre ninguna penalización en la puntuación obtenida.

3.3. Framework utilizado

Esta versión es una copia muy completa del original, e incluye multitud de métodos para acceder a los elementos del juego, aunque se hayan tenido que implementar algunos nuevos, necesarios para implementar algunas ideas presentadas en este proyecto.

Aunque es una versión muy aproximada a la original, tiene sus particularidades, que se comentan a continuación:

- Tiene un contador de tiempo que cuenta “tics” de reloj. En esta versión, si el jugador aguanta 3000 tics en el mismo laberinto (no necesariamente sin que le coman alguna vez, con que no agote todas sus vidas es suficiente), el juego salta al siguiente nivel, y le da al jugador tantos puntos como los que acumulan las píldoras que quedaban en el laberinto anterior.
- El mapa está dividido en nodos, de forma que cuando el jugador realiza un movimiento, en realidad está moviéndose de un nodo del mapa a otro, y no existen movimientos intermedios que se puedan realizar entre ellos. Para hacernos una idea de la escala que se maneja, entre cada dos píldoras contiguas hay cuatro nodos del laberinto, en los que el jugador puede cambiar de dirección si lo desea.
- En esta versión se dispone de varias IA para los fantasmas, contra las que podemos enfrentarnos. La IA básica que viene por defecto, por ejemplo, consiste en fantasmas

que tienen dos estados. En uno de ellos persiguen al jugador por el laberinto, y en el otro, se dedican a patrullar por alguna zona concreta del nivel.

A continuación se detallará cómo está estructurado el código del videojuego y posteriormente las clases nuevas que se han implementado y cómo funcionan.

Las clases principales que hay que estudiar para entender cómo funciona el simulador son las siguientes:

- **Game.java:** Esta interfaz define la relación que tendrán los bots con el juego. En ella están definidos todos los métodos y variables necesarios para acceder a los datos de la partida. Se puede obtener información de fantasmas, píldoras, distancias o rutas dentro del nivel.
- **G.java:** Esta clase implementa la interfaz anterior. En ella están programados todos los métodos necesarios para la ejecución del juego, y también para obtener información sobre él. Ejemplos de estos son métodos que devuelven la posición de los fantasmas, los vecinos de un nodo, o las píldoras que quedan en el laberinto.
- **PacManSimulator.java:** En esta clase está el *main* del juego, desde el que se van ejecutando los episodios de la simulación.
- **SimulatorConfig.java:** Esta clase determina los parámetros bajo los que se ejecuta el juego. Se puede cambiar, por ejemplo, la velocidad del juego, o si la partida se muestra de forma gráfica o no.
- **PacManHijackController.java:** De esta clase tendrán que heredar todos los bots que se programen. Esta implementa la interfaz PacManControllerBase situada en el mismo package, y contiene todos los métodos necesarios para operar con los bots.

Llegados a este punto, una vez vista toda la base teórica, y el funcionamiento del juego, ya tenemos todo lo necesario para comenzar a hablar de la implementación de algoritmos de aprendizaje por refuerzo, y razonamiento basado en casos en Ms. Pac-Man.

Capítulo 4

Implementando bots para Ms. Pac-Man

Introducción

En este capítulo se verá todo el proceso de diseño e implementación de un bot para el videojuego Ms. Pac-Man.

Se comienza viendo las clases que se van a implementar, y posteriormente se estudia en profundidad cómo se pueden aplicar las técnicas de aprendizaje por refuerzo vistas anteriormente, para después combinarlas con métodos de razonamiento basado en casos.

4.1. Estructura del bot

Para implementar un bot que utilice las técnicas de aprendizaje por refuerzo y razonamiento basado en casos, se han construido las siguientes clases principales:

- **Q_learn.java:** En esta clase está toda la información y métodos relacionados con el aprendizaje y el razonamiento, desde el propio algoritmo Q-learning hasta los métodos necesarios para gestionar la base de casos, como los métodos de búsqueda, comparación de casos, estadísticas de las simulaciones, etc.

- **QLearnDirPacMan.java:** Aquí es donde se implementa el bot que utilizará la información de la clase anterior para jugar y aprender.

- **GameState.java:** Esta clase es fundamental, pues contiene los datos que el agente utiliza como *estado* del juego, es decir, la información que usa para decidir en qué tipo de situación se encuentra, y por tanto cómo actuará en cada situación.

- **SearchThread.java:** Aquí se implementa la búsqueda con hilos, como método para optimizar el proceso de aprendizaje aprovechando las capacidades multicore del procesador. Cuando la base de casos llega a un cierto tamaño, la búsqueda pasa a realizarse mediante hilos que trabajan de forma paralela.

4.2. Modelando el estado de juego

Para que el agente pueda aprender a jugar o interactuar con su entorno, lo primero que hay que hacer es darle una representación válida del mismo. Esto es una tarea complicada, y de las más importantes, pues el juego tiene muchas características diferentes, y hay que elegir las que mejor informen al agente de lo que ocurre a su alrededor.

Aparte de seleccionar las características que más información den al agente, hay que intentar que estas sean mínimas, es decir, que la representación del estado del juego de que dispone el agente sea lo más pequeña posible, para poder mantener la base de estados dentro de un tamaño razonable y que así sea más manejable.

Para esta tarea tratemos de imaginar cómo juega un humano a Pac-Man, en qué cosas del juego se fija para tomar decisiones, y, en definitiva, qué elementos del juego son importantes para elegir moverse a un sitio u otro.

Veamos ahora qué elementos del juego podríamos seleccionar para representar el estado del juego:

- El bot debe tener información de algún tipo acerca de las píldoras que hay en el laberinto. Guardar la localización de todas las que quedan nos daría un estado demasiado grande, y además muchas de ellas no son necesarias a la hora de tomar una buena decisión. Por tanto, se llega a la conclusión de que la mejor decisión es que tenga información sobre la distancia a la que se encuentra una píldora en cada dirección (la más cercana).

- Los fantasmas juegan un papel clave en una partida, pues determinan cuándo concluye, al comer al jugador, por lo que este debe tener siempre un ojo puesto en ellos. Esta información debemos dársela a nuestro bot, en forma de datos. Si bien decirle la posición de los fantasmas en todo momento es una opción válida, y a priori buena, al haber tantas posiciones posibles se produciría un número de estados enorme en cada partida, de manera que la probabilidad de volver a recabar información sobre ellos sería muy pequeña. En vez de esto, podemos imaginarlo como en el caso anterior y decirle a nuestro agente a qué distancia tiene un fantasma en cada una de las direcciones.

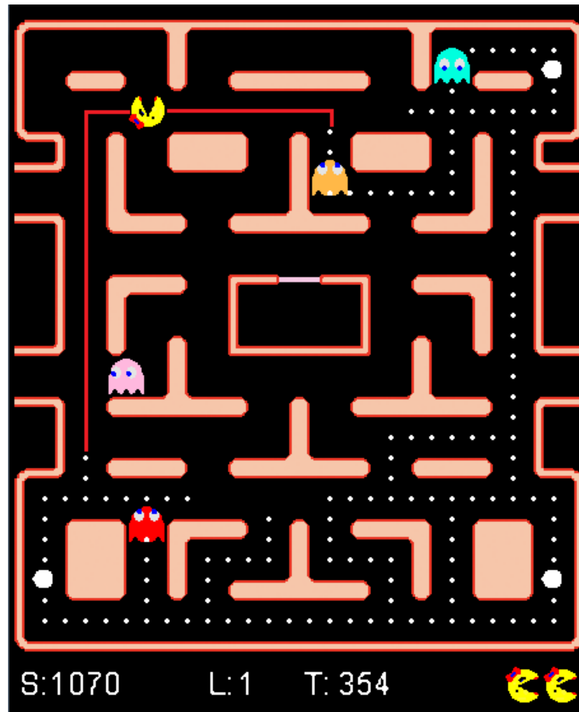

Figura 4.1 Píldoras cercanas a Pac-Man.

Esto ya nos valdría para que nuestro bot supiera en qué entorno se encuentra, y aprendiera a moverse por él, en base a los dos elementos principales del laberinto, que son las píldoras y los fantasmas. A partir de aquí todo lo que añadamos será para mejorar ciertos tipos de comportamientos, y con ello ayudarle a conseguir una mayor puntuación. Un elemento clave para esto son las cuatro *power pills* o píldoras especiales repartidas por el mapa. Estas cambian a los fantasmas de estado de forma que ahora pueden ser devorados por Pac-Man y con ello recibirá puntos extra. Sería por tanto muy útil que el bot dispusiera de información acerca de estos elementos:

- Como en el caso de las píldoras normales, es necesario conocer a qué distancia está la píldora especial más cercana en cada dirección.
- Uno podría pensar que con los datos anteriores de los fantasmas podría ser suficiente, simplemente sabiendo si se ha comido una píldora recientemente, pero el problema está en que pueden coexistir fantasmas comestibles y no comestibles a la vez en el laberinto, si comemos un fantasma, en cuyo caso este vuelve a su casilla de salida, en estado de ataque de nuevo, mientras el resto sigue en estado comestible. Por tanto necesitamos datos de a qué distancia está el fantasma comestible más cercano, en cada dirección.

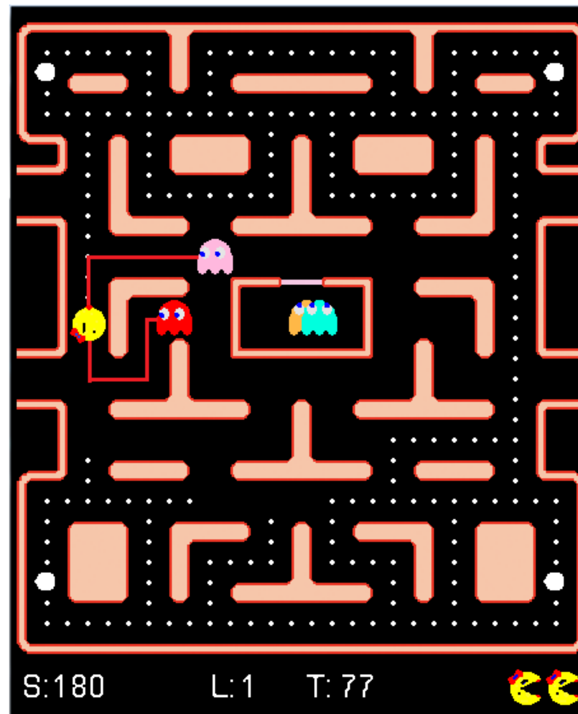

Figura 4.2 Fantasmas cercanos a Pac-Man.

- Aunque en cierta dirección tenga un fantasma relativamente cerca que le pueda comer, un jugador humano puede detectar si hay rutas de escape en dicha dirección, por las que pueda huir sin ser devorado. Una manera simple de verlo es observar que en el mapa hay ciertas posiciones que son intersección de dos o más calles del laberinto, y en las que Pac-Man puede torcer en una u otra dirección. Por tanto, la forma más sencilla de que el agente sepa si puede existir una ruta de escape o no en una dirección es saber si tiene una intersección cerca, por ejemplo.

Teniendo estos datos para cada dirección, nos queda un espacio de estados posibles increíblemente grande. A partir de ahora comienza el reto de aprender a gestionar esta información para que el agente pueda razonar de forma correcta.

Un primer paso que podríamos dar, con el fin de reducir el espacio de estados el máximo posible, sería discretizar las distancias en unos pocos valores, para así agrupar ya de antemano distancias parecidas en unas pocas clases. Estas distancias se miden en número de nodos o casillas del laberinto que hay entre los dos elementos. Así, cuando se genera un estado, se establece que las distancias pueden tomar los valores *MUY CERCA* $(0 \leq d < 4)$, *CERCA* $(4 \leq d < 49)$, *MEDIO* $(49 \leq d < 225)$ y *LEJOS* $(225 \leq d)$. Adicionalmente, se añade el valor *PARED*, para aquellos estados en que una dirección no sea accesible.

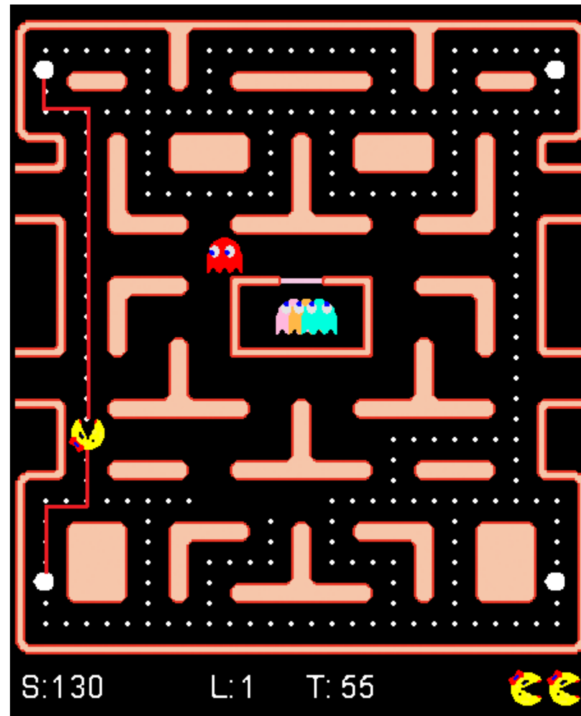

Figura 4.3 Píldoras especiales cercanas a Pac-Man.

Con esta representación, hemos reducido el espacio de estados a “solo” $5^{20} = 95367431640625$ posibilidades. Como vemos, siguen siendo demasiadas, pero en las siguientes secciones se verán formas de gestionar esto. Además, por el diseño del juego (por ejemplo, muchos nodos se encuentran en pasillos, y por tanto solo tienen dos direcciones accesibles), muchos de estos estados no son siquiera posibles.

Estos son, por tanto, algunos ejemplos de información que pueden formar parte del estado del juego. Si se incluyeran todos a la vez, aún habiendo discretizado las distancias, el espacio de estados posibles sería enorme, lo que dificulta mucho la tarea de operar con una tabla Q, pues el tamaño de esta sería demasiado grande. En las siguientes secciones se verán formas de combinar las ideas de la tabla Q con el razonamiento basado en casos, para poder operar con estados de esta complejidad.

En el siguiente capítulo se ven algunos experimentos en los que se utiliza aprendizaje por refuerzo con estados más sencillos, en los que se utilizan solo algunas de estas características, y su posterior mejora al añadir más información a los estados junto con CBR.

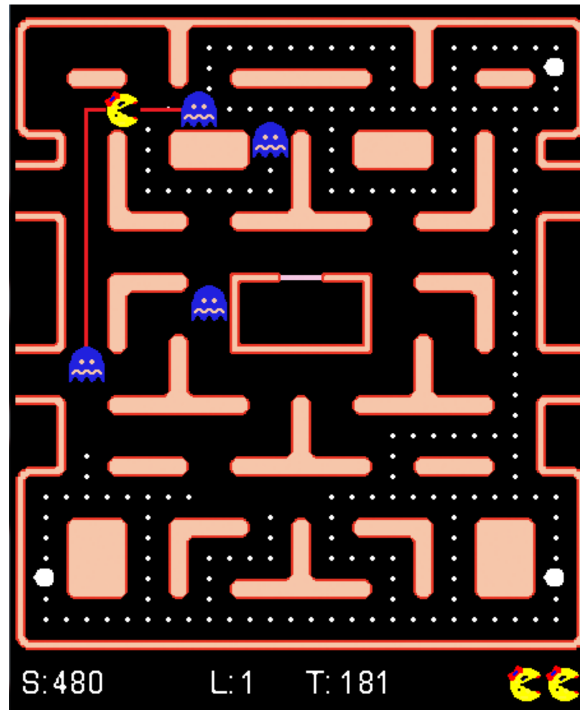

Figura 4.4 Fantasmas comestibles cercanos a Pac-Man.

4.3. Recompensas

Determinar las recompensas que obtiene el agente con cada acción es clave para su aprendizaje, pues marca cómo de buenas serán unas acciones frente a otras y por lo tanto decide qué tipo de comportamientos decidirá potenciar a la larga.

Para estudiar qué recompensas se pueden otorgar en base a las acciones, y dado que el objetivo es maximizar la puntuación total obtenida, habría que pararse a pensar qué elementos dan puntos, o cómo se mide lo buena o mala que ha sido una acción en cada momento.

Ya se vio en la descripción del juego que los elementos que dan puntos son las píldoras, píldoras especiales, y los fantasmas en estado comestible. En base a esto podemos determinar que la recompensa de pasar de un caso ***c*** a un caso ***c'*** sea

$$r = Score_{c'} - Score_c$$

También se vio que una de las características ocultas de esta versión de Ms. Pac-Man determina que el jugador pasará de nivel (y ganará una puntuación equivalente a comerse todas las píldoras restantes) si sobrevive un cierto número de turnos. Esto hace que mantenerse vivo sea un elemento importante en lo que a recompensas se refiere, pues estar vivo siempre va a ser positivo para el agente, ya sea para sobrevivir este tiempo que le permite pasar el

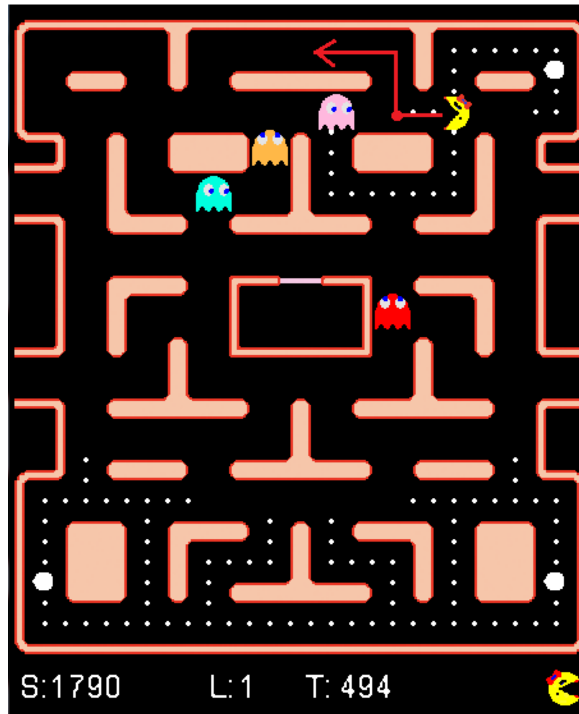

Figura 4.5 Intersecciones cercanas a Pac-Man.

nivel, o para seguir comiendo píldoras o fantasmas y continuar ganando puntos. Siguiendo la idea que se utilizó con las puntuaciones, una recompensa válida al pasar de un caso al siguiente después de una acción podría ser:

$$r = time_{c'} - time_c$$

y combinando las dos se obtiene otra posible función de recompensa:

$$r = Score_{c'} - Score_c + time_{c'} - time_c$$

Todas estas serán puestas a prueba posteriormente, estudiando los resultados obtenidos y viendo qué acciones favorecen más unas u otras.

4.4. Espacio de acciones

Para poder implementar Q-learning correctamente, debemos tener un espacio de acciones bien definido, para saber qué es lo que puede y no puede hacer el agente en cada situación.

En este caso las acciones serán moverse en cada una de las cuatro direcciones principales, es decir, moverse arriba, abajo, a la izquierda o a la derecha. Habrá muchas situaciones en

las que no tenga todas las direcciones disponibles, por ejemplo cuando se encuentre en un pasillo.

4.5. Q-learning y Pac-Man

Ya tenemos la información básica que regirá el proceso de aprendizaje, ahora solo hay que ver cómo almacenarla y manejarla correctamente, y cómo opera el algoritmo Q-Learning en este videojuego.

4.5.1. Memoria de estados

Para que el agente pueda aprender, necesita tener a mano todos los estados que haya visitado anteriormente, para poder detectar en cual de ellos se encuentra a medida que va jugando y, en caso de encontrarse en una situación nueva, registrarla para utilizarla posteriormente.

En la implementación, estos estados se guardarán en forma de lista a medida que se vayan descubriendo.

4.5.2. La tabla Q

Como se vió cuando se hablaba de Q-learning, el agente aprende modificando los valores de la función $\boldsymbol{Q}$ (los valores que hay almacenados en la tabla Q), de forma que cuando se realiza la acción a desde el estado s, es el valor $Q(s,a)$ el que se ve modificado de acuerdo a la ecuación del algoritmo.

En el caso de Pac-Man, desde cada estado solo existen cuatro acciones posibles (ir en cada una de las cuatro direcciones), con lo que la tabla quedaría como se ve a continuación.

Estado	Valor arriba	Valor derecha	Valor abajo	Valor izquierda
1	0.84	54.35	1.2	17.85
2	3.5	2.1	15.4	2.365
3	67.1	4.6	2.6	19.5
...	...	...	...	...

Cuadro 4.1 Ejemplo de tabla Q en Pac-Man

Como en el caso de Pac-Man las recompensas son positivas, las acciones son mejores cuanto mayor es su valor correspondiente en la tabla Q. Así, en la tabla de ejemplo, desde el

estado 1 la mejor acción posible para el bot es irse a la derecha, desde el estado 2, irse hacia abajo, etc.

4.5.3. Agente

Ya sabemos cómo está organizada la información que el agente utilizará para decidir qué hacer, ahora veamos cómo decide el agente qué dirección debe escoger en cada momento, es decir, la ***política*** que sigue.

Como también se vio en el capítulo correspondiente, esta política debe basarse de alguna manera en la información de la tabla Q, con lo que también se va modificando con cada episodio, para eventualmente llegar a ser una política óptima.

En este caso, y dado que las recompensas por cada acción son siempre positivas, se establece que la política del agente sea realizar siempre la acción con mayor valor en la tabla Q, salvo algún movimiento aleatorio que ocurre con probabilidad ε.

Esta política, como ya se vio cuando se explicaba el algoritmo Q-learning, se conoce como ***ε-greedy*** y su objetivo es dejar un cierto margen a la exploración de estados. Con ese movimiento aleatorio de vez en cuando se pretende que el agente explore el mayor número de estados posibles y pueda descubrir más alternativas. De no ser así, podría quedarse atascado en un valor que sea óptimo "localmente", privándose así de encontrar soluciones mejores.

4.5.4. Algoritmo Q-Learning

Recordamos el algoritmo Q-learning para mayor comodidad del lector:

Algoritmo 2: Algoritmo Q-learning en Pac-Man

```
1  Inicializa Q(s,a) arbitrariamente;
2  repetir
3  |   Inicializa estado s;
4  |   repetir
5  |   |   Busca estado s en la memoria de estados;
6  |   |   Escoger la dirección a siguiendo la política del agente Pac-Man;
7  |   |   Moverse en dirección a, y luego observar r, s';
8  |   |   Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)];
9  |   |   s ← s';
10 |   hasta que s es terminal;
11 hasta que no haya más episodios;
```

Hay que hacer ciertas aclaraciones para asegurar que se entiende su funcionamiento. La primera de todas es que los valores de la tabla Q unicamente son actualizados cuando

se cambia de estado, es decir, si al tomar una acción, el agente se mantiene en el mismo estado, la tabla Q no se actualiza, sino que la recompensa se acumula, de forma que se suman todas las recompensas obtenidas mientras se permanece en un estado, y es cuando se pasa al siguiente el momento de modificar la tabla.

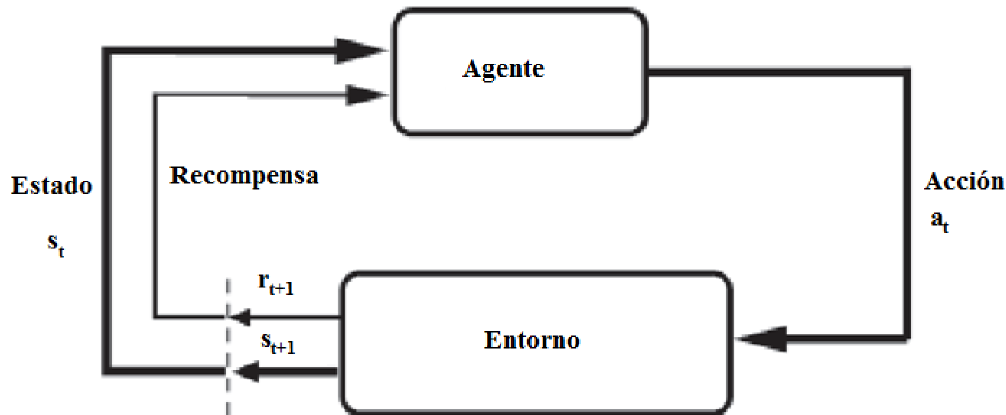

Figura 4.6 Ciclo Q-learning

4.6. Mezclando RL y CBR

Como se ha visto ya, es necesario encontrar un equilibrio entre la cantidad de información que le proporcionamos al agente para operar, y el tamaño de nuestra memoria de estados, es decir, gestionar la información que contiene un estado, cuándo este debe entrar en la memoria, o si alguno debería salir, si es necesario.

Para esto, trataremos de combinar el aprendizaje por refuerzo que proporciona el algoritmo Q-learning con la gestión de la información que da el razonamiento basado en casos.

4.6.1. El Caso y la Base de Casos

Hablando de razonamiento basado en casos, debemos hablar entonces de la diferencia entre lo que antes llamábamos *estado* en Q-learning, y lo que es un *caso* en CBR.

Si bien un estado es la representación del juego en una situción determinada, un caso es también esa información, pero junto con la solución a dicha situación.

En el caso de nuestro videojuego, un caso podría verse como un estado de juego al que le añadimos la información de la tabla Q correspondiente a ese estado, pues es esta información de la tabla la que determina cómo actuaremos en esa situación.

Una de las ventajas de usar CBR es que los casos son más específicos que los estados clásicos que describíamos anteriormente, ya que ahora tendremos métodos para evitar que el tamaño de la base de casos se vuelva demasiado grande tras simular pocos episodios.

Estos métodos consistirán en no añadir cada situación nueva como un nuevo caso en la memoria, sino que podamos reutilizar experiencias pasadas que no sean exactamente iguales, pero que podamos adaptar a la actual.

Para ello, por tanto, debemos establecer cómo se pueden comparar dos casos, es decir, establecer alguna *medida de similitud* entre casos.

Por todo esto, ahora podemos añadir más información a un caso de la que tenía el estado que hemos utilizado en las secciones previas. Cuando hablamos de casos, podemos determinar que un caso almacena, para cada dirección, la siguiente información sobre entorno del agente:

- Distancia a la píldora más cercana.
- Distancia a la píldora especial más cercana.
- Distancia al fantasma más cercano.
- Distancia al fantasma comestible más cercano.
- Distancia a la intersección más cercana.

Por ejemplo, en la siguiente situación, el agente se encuentra como en la imagen:

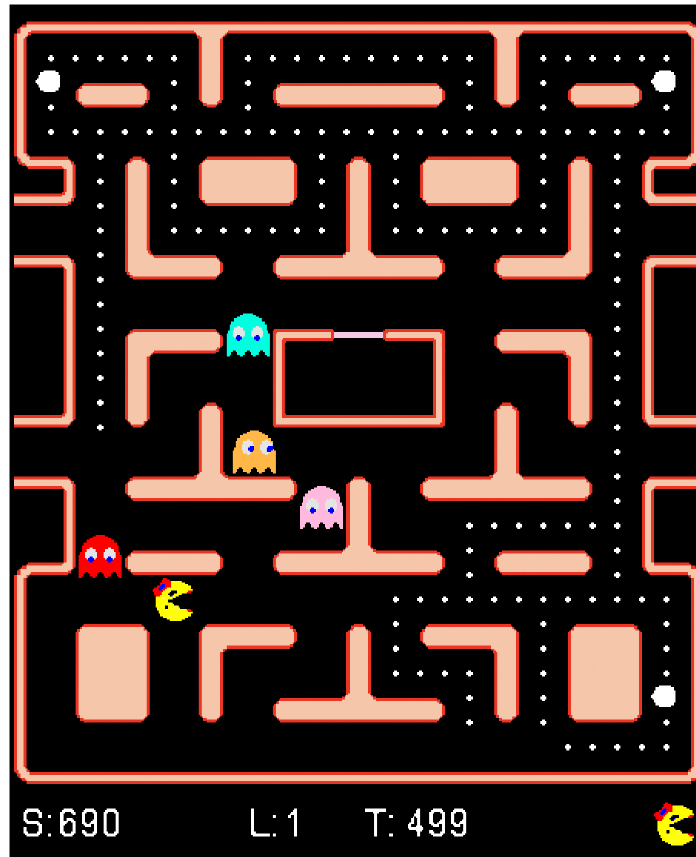

Figura 4.7 Ejemplo de caso

y esta situación la tendrá que modelar como un caso, adaptando los datos que obtiene del juego a los que hemos elegido que representen dicho caso en nuestro framework.

Las distancias a cada uno de los elementos se agruparán en vectores cuyas componentes se refieren a distancias en distintas direcciones de la siguiente manera: [ARRIBA, DERECHA, ABAJO, IZQUIERDA].

Entonces, la imagen de arriba se traducirá en el siguiente caso:

- Píldoras: [PARED,MED,MED,MED]
- Fantasmas: [PARED, MED, MED, CERCA]
- Píldoras especiales: [PARED, LEJOS, LEJOS, LEJOS, LEJOS]
- Fantasmas comestibles: [PARED, PARED, PARED, PARED]. El valor que el agente percibe cuando no existen elementos como el que se buscan es el mismo que cuando hay una pared.
- Intersecciones: [PARED, CERCA, CERCA, CERCA]

4.6.2. Medidas de distancia y similitud

Supongamos que tenemos un caso ***c***, ¿cuál es el caso más parecido a c de entre los que tenemos almacenados en la base de casos? Determinar esto es fundamental cuando se construye un agente que razona mediante casos.

Además, esta respuesta puede no ser tan trivial como pueda parecer, y dependerá de lo que se entienda por "similar", del entorno en el que se encuentre el agente, y de la información que contenga el caso.

Debe definirse entonces una función de similitud ***sim*** entre dos casos ***c*** y ***c'***, que determine qué significa que dos casos sean parecidos, y cuantifique esta semejanza. Para que esta función sea efectiva vamos a buscar que cumpla ciertos requisitos:

- $0 \leq sim(c,c') \leq 1$
- $sim(c,c) = 1$ (reflexiva)
- $sim(c,c') = sim(c',c)$ (simétrica)

La similitud entre dos casos será calculada en función a una medida de distancia que definamos entre ellos. Esta función servirá para medir la diferencia entre los datos de un caso al otro. También esta función debe definirse para que cumpla un par de requisitos:

- $d(c,c') \geq 0$
- $d(c,c) = 0$

Un ejemplo de distancia podría ser la distancia euclídea, que vendría dada, siendo f_i y f'_i las características de *c* y *c'* respectivamente, por:

$$d_E(c,c') = \sqrt{\sum_{i=1}^{n} (f_i - f'_i)^2}$$

Veamos ya cómo aplicar todo esto a Pac-Man. Dados dos casos, que sabemos están representados cada uno por 20 enteros, calcular la distancia entre dos de ellos se antoja sencillo. Solo hay que asignar valores numéricos a las distancias discretizadas, a saber:

- -1: PARED
- 1: MUY CERCA
- 2: CERCA

- 3: MEDIO
- 4: LEJOS

También, y como medida de optimización, al ser esta operación muy repetida en cada simulación, podemos quitar la raíz cuadrada y la elevación al cuadrado de los cálculos, para aligerar al máximo operaciones costosas. Tenemos por tanto una función propia de distancia, que es:

$$d_E(c,c') = \sum_{i=1}^{n} |c_i - c'_i|$$

Para simplificar el cálculo de distancias, empezamos utilizando como función de distancia la siguiente (aunque posteriormente podemos utilizar la que acabamos de explicar, ya que realiza cálculos más precisos):

$$d_E(c,c') = \sum_{i=1}^{n} j_i$$

donde $j_i = 1$ si c_i y c'_i son iguales, y 0 en caso contrario.

Sabiendo esto, en vez de calcular la distancia usando todos los datos de golpe, los agrupamos en categorías, con el fin de poder disponer de distintas funciones de similitud, en función de a qué información queramos que nuestro agente le de más importancia a la hora de aprender.

Así, calculamos por separado:

- La distancia entre los vectores de datos referentes a píldoras (*pills*), d_p.
- La distancia entre los vectores de datos referentes a píldoras especiales (*power pills*), d_{pp}.
- La distancia entre los vectores de datos referentes a fantasmas (*ghosts*), d_g.
- La distancia entre los vectores de datos referentes a fantasmas comestibles (*edible ghosts*), d_{eg}.
- La distancia entre los vectores de datos referentes a intersecciones, d_i.

A partir de aquí es donde definimos la función de similitud que utilizará el agente para diferenciar unos casos de otros. Mediante un sistema de pesos podremos ponderar las distancias a nuestro antojo, para, como ya se dijo, que el agente de más importancia a unos elementos que a otros. Esto proporciona un abanico de posibilidades muy grande, y sería interesante estudiar qué tipo de funciones de similitud funcionan mejor, o métodos para obtener una función de similitud óptima.

Si asignamos los pesos $w_1 \cdots w_5$, tales que $\sum_{i=1}^{5} w_i = 1$, la función de similitud queda:

$$s = w_1 d_p + w_2 d_g + w_3 d_{pp} + w_4 d_{eg} + w_5 d_i$$

Pero esta función no nos da un valor entre 0 y 1, como pedíamos al principio, sino que, al sumar 1 todos los pesos, nos dará un valor entre 0 y el máximo de distancia entre dos de los vectores de cuatro componentes, que en este caso será una distancia de 4, según hemos definido todos los parámetros. Que la similitud venga dada entre 0 y 1 será fundamental para poder adaptar las soluciones como se verá más adelante.

Se normaliza entonces la similitud, dividiendo esta cantidad entre la distancia máxima entre vectores, MD, y nos queda que la similitud normalizada es:

$$s_N = (1 - s/MD)$$

Por ejemplo, supongamos que tenemos los dos siguientes casos:

Caso	Píldoras	Fantasmas	Píldoras esp.	Fantasmas comest.	Intersecciones
Caso 1	[-1,3,-1,3]	[-1,3,-1,2]	[-1,4,-1,4]	[-1,-1,-1,-1]	[-1,1,-1,2]
Caso 2	[-1,3,3,3]	[-1,3,3,2]	[-1,4,4,4]	[-1,-1,-1,-1]	[-1,2,2,2]

Cuadro 4.2 Ejemplo de similitud de casos

Y un vector de pesos $w = (0.05, 0.875, 0.025, 0.025, 0.025)$. Lo primero que habrá que hacer será calcular las distancias entre cada uno de los grupos. Así, tendremos que

$$d_p = 1, d_g = 1, d_p p = 1, d_e g = 0, d_i = 2$$

Lo siguiente calcular la similitud a partir de estas distancias y los pesos:

$$s = 0.05 * 1 + 0.875 * 1 + 0.025 * 1 + 0.025 * 1 + 0.025 * 2 = 1.025$$

Y entonces la similitud normalizada será:

$$s_N = (1 - 1.025/4) = 0.74375$$

4.6.3. Recuperación de casos y nuevos casos

Ya sabemos cómo pueden compararse dos casos y cómo medir por tanto la similitud entre ellos. La función descrita en el apartado anterior devuelve un número entre 0 y 1, donde 1 representa que los dos casos son iguales, y 0 que son lo más diferentes posible.

La siguiente tarea entonces es ver cómo se recuperan los casos de la base de casos, y cuándo debemos añadir un nuevo caso a la misma.

Cuando el agente se encuentra en una nueva situación, después de realizar cierta acción, toma todos los valores necesarios del juego y genera un caso (discretizando los valores de distancias). Acto seguido comprueba si el caso actual es igual al caso en el que se encontraba antes de realizar la acción, y, de ser así, no hace más que acumular la recompensa recibida hasta que vuelva a realizar otra acción. En caso contrario, significa que el agente se encuentra en un caso distinto, y tendrá que revisar la memoria para ver si coincide, o si se parece lo suficiente, a alguno de los casos que tiene almacenados.

El proceso entonces es ir a la base de casos y buscar aquel con el que tenga mayor similitud, para ver si podemos adaptar la solución de ese al caso actual. Cuando lo encuentre, aunque sea el caso más similar en memoria, puede que este no ofrezca una semejanza suficiente como para que consideremos válido para ayudar en el caso actual. Si el agente se ve en esta situación, en vez de tomar el caso más similar aunque no sea válido, lo que hace es añadir el caso actual a la base de casos, para poder reutilizarlo más adelante. Esta decisión la toma agente en base a un umbral predefinido, que mide cómo de diferentes tienen que ser los casos nuevos de los que se encuentran en memoria para ser añadidos a la base de casos.

La búsqueda del caso más similar se realiza de forma lineal en la base de casos, comparando con el caso actual hasta que se encuentra el más similar. En capítulos posteriores se verán formas diferentes de hacer eso, con el fin de optimizar el tiempo y los recursos en el proceso de recuperación de casos.

4.6.4. Adaptación de soluciones

Una vez que hemos recuperado un caso apto por ser suficientemente similar al caso actuar, lo siguiente que hay que hacer es adaptar la solución de uno a la del otro, pues en la mayoría de ocasiones la solución no puede aplicarse directamente, sino que hay que modificarla de alguna manera para que tenga un efecto lo más parecido posible.

En el caso del Pac-Man, adaptar la solución significa dos cosas:

- Comprobar si mejor dirección posible que se puede tomar en un caso, se puede tomar en el otro. Si esto no es posible, el agente debe tomar la mejor dirección entre aquellas a las que se puede dirigir desde su posición.

- Modificar la recompensa que se da al aplicar la solución. Dado que el caso no es exactamente el mismo, la recompensa que se transmite al caso recuperado para modificar los datos que ha aprendido tampoco debe ser la recompensa obtenida completa, sino que debe reducirse de alguna manera en base a la medida de similitud que separa esos casos, para simbolizar el aprendizaje que se produce desde casos similares pero no exactamente iguales. Aquí entra la importancia de que la función de similitud debía estar normalizada entre 0 y 1, como comentamos anteriormente. De esta manera, simplemente multiplicando en la ecuación la recompensa por la similitud normalizada podemos tener el aprendizaje adaptado cuando razonamos con casos.

El algoritmo CBR queda entonces de la siguiente manera:

Algoritmo 3: Algoritmo RL/CBR en Pac-Man

```
1  Inicializar base de casos;
2  repetir
3  |  Inicializar caso c;
4  |  repetir
5  |  |  Buscar el caso c' más similar en la base de casos. Calcular la similitud s(c,c') (
   |  |    0 ≤ s ≤ 1 );
6  |  |  si c y c' son suficientemente similares entonces
7  |  |  |  Elegir dirección a a partir del caso c';
8  |  |  |  Tomar dirección a, y observar recompensa r, y siguiente caso c'';
9  |  |  |  Q(c',a) ← Q(c',a) + α[s*r + γ max_a' Q(c'',a') − Q(c',a)];
10 |  |  en otro caso
11 |  |  |  Añade c a la base de casos;
12 |  |  fin
13 |  |  c ← c'';
14 |  hasta que no haya más pasos en el episodio;
15 hasta que no haya más episodios;
```

4.7. Gestión de la Base de Casos

A medida que se simulan episodios, el agente va pasando por muchas situaciones distintas, las cuales irá almacenando o no según el criterio que le hayamos dado. Es importante mantener un equilibrio entre la cantidad de información que le damos a un caso, y el umbral que haya para añadir un nuevo caso a la memoria, pues de nada sirve añadir muchos casos a la memoria si luego, al ser tan específicos, los vamos a recuperar muy pocas veces cada uno.

Se trata de construir un entorno que permita almacenar casos que proporcionen información valiosa al agente, y que además se puedan reutilizar muchas veces. Esto último es para que se cumpla uno de los requisitos de convergencia del algoritmo Q-learning utilizado, que es que las soluciones de casos deben ser continuamente actualizadas, y esto solo se puede conseguir recuperándolos muchas veces durante las simulaciones de episodios.

Aun así, habrá muchas situaciones únicas por las que pasará el Pac-Man en las que pocas veces se volverá a encontrar y que, por tanto, podemos considerar descartar de nuestra memoria. Ocupan espacio, y hacen perder tiempo al agente durante las búsquedas. Por tanto debemos buscar medios para gestionar la base de casos de forma correcta.

Viendo la cantidad de casos posibles que se pueden dar, la primera idea sería poner un límite al número de casos que almacenamos en la memoria del agente. Aunque esto conlleva limitar el aprendizaje de dicho agente, el objetivo ahora es buscar cómo aprovechar ese límite en la base de casos para llenarla con estados que de verdad ayuden al agente a ganar puntos, y descartar el resto.

En nuestro caso se toma la opción de que cuando se llega a ese límite, el agente borra un porcentaje predeterminado de casos siguiendo el siguiente procedimiento:

- Se selecciona un caso al azar, y se busca el caso más similar en la base de casos. Si ambos casos tienen la misma solución, se elimina de la base de casos aquel que haya sido recuperado menos veces.

Algoritmo 4: Algoritmo de borrado de casos

1 **repetir**
2   Seleccionar un caso **c** al azar;
3   Buscar el caso *c'* más similar en la base de casos. **si** *c y c' hacen que PacMan tome la misma dirección* **entonces**
4   Borra el caso que haya sido recuperado menos veces;
5   **fin**
6 **hasta que** *10 % de casos borrados*;

4.8. Mejorando el rendimiento

Ya hemos visto todo lo que necesita el bot para aprender. Manejar toda esta información durante miles de episodios cuesta muchos recursos, y tiempo. Preocuparse por acelerar este proceso de aprendizaje es una tarea importante si queremos que nuestro agente acabe su labor en un tiempo razonable.

Algunas de las técnicas que se ven a continuación ya han sido introducidas en secciones anteriores, como parte natural del desarrollo, mientras que otras se incluyeron posteriormente, con el fin de optimizar las simulaciones.

Veamos las medidas que se han tomado con este objetivo:

- **Discretizar ditancias:** Dado que hay muchos nodos en cada laberinto, calcular distancias exactas lleva a muchos valores posibles para cada uno de los datos que forman un caso. Esto hace que la base de casos crezca muy rápido, y dificulta la tarea de que a cada función lleguen siempre valores consistentes y correctos. Para ahorrarse esto, y sobre todo para reducir el tamaño y crecimiento de la base de casos, se realiza una discretización de las distancias para incluirlas en la categoría que les corresponda. Mantener la base de casos en un tamaño razonable es importante en cuestiones de tiempo. Durante la simulación de miles de episodios se aprecia claramente que los primeros se simulan rápidamente, pero a medida que la base de casos crece y las búsquedas requieren más esfuerzo, el proceso se ralentiza de forma considerable.

- **Eliminar operaciones costosas:** El aprendizaje requiere de muchos cálculos que deben realizarse lo más rápido posible para que el proceso se complete en un tiempo tan corto como se pueda. Comparar casos, actualizar las soluciones y obtener información del juego, son solo algunas de estas operaciones, que deben ser optimizadas al máximo. Ya se han comentado algunas, como por ejemplo quitar operaciones de las funciones que calculan la distancia entre casos, para hacer que esta función sea lo más sencilla y rápida posible.

- **Limite en la base de casos:** Buscar en la base de casos requiere muchos cálculos pues debe comparar el caso actual con todos los que hay almacenados para ver cuál es el más similar. Una forma de mantener esto dentro de un tiempo razonable es limitar el número de casos totales almacenados, para que nunca lleguemos a una situación en que haya que buscar entre tantos casos que la lentitud entorpezca de verdad este proceso.

- **Búsqueda con hilos:** Con el mismo objetivo que en el apartado anterior, una manera de acelerar la búsqueda es realizarla utilizando varios hilos que realicen dicha búsqueda en paralelo. Con esto se pretende aprovechar todos los núcleos que tenga el procesador, asociando a cada hilo una sección de la base de casos y luego comparando los casos que devuelva cada uno de ellos.

- **Precalcular distancias:** Para acelerar el paso de generar un caso a partir del estado del juego actual, se han precalculado las distancias de cada nodo a todos los demás en

cada dirección, para que el bot solo tenga que realizar ese esfuerzo una vez, y no tenga que calcular estas distancias según va avanzando por el laberinto una y otra vez.

Capítulo 5

Experimentos y resultados

Introducción

En este capítulo se presentan los experimentos realizados, tanto utilizando técnicas de aprendizaje por refuerzo exclusivamente, como su posterior combinación con razonamiento basado en casos.

5.1. Experimentos y resultados de RL

En estos primeros experimentos se ve el uso del algoritmo Q-learning en Pac-Man cuando le proporcionamos al agente distintos tipos de información, las limitaciones que presenta y los resultados que se pueden obtener con él.

Nuestro bot tomará decisiones cada vez que llegue a una intersección, momento en que mirará el estado del juego, verá en qué situación se encuentra, y en base a los conocimientos aprendidos tomará la decisión de moverse en una dirección u otra. También, en aquellos experimentos en que se le proporcione información acerca de los fantasmas, el bot también tomará decisiones en el momento en que perciba a un fantasma demasiado cerca de él (cuando la distancia entra en el rango *MUY CERCA*), momento en el cual puede decidir darse la vuelta aunque no se encuentre en una intersección. Competirá en todo momento contra un bot que realiza movimientos aleatorios en cada intersección.

Probaremos ahora el algoritmo de aprendizaje y veremos cómo se desenvuelve cuando damos al agente distintos tipos de información, estudiando los comportamientos que desarrolla el bot.

El número de partidas simuladas será 10000, y será el mismo para todos los experimentos. También, como el agente escoge movimientos aleatorios con una probabilidad ε,

establecemos este en un valor inicial del 100 %, que irá decreciendo conforme avanzan las simulaciones, lo que terminará favoreciendo la exploración de estados y con ello un mejor aprendizaje final de nuestro bot.

5.1.1. Experimento 1

En este primer experimento, el bot solo recibe información sobre las píldoras, y establecemos que la recompensa que recibe por cada acción esté basada en la puntuación. Esto quiere decir que la recompensa por realizar una acción en cierto estado será la puntuación que el agente obtenga en el siguiente estado. El objetivo de esto es que el agente aprenda qué acciones van a reportarle más puntos en cada momento.

Resultados: Se observa en el bot que aprende a perseguir las píldoras para comérselas, pero al no tener constancia de dónde están los fantasmas, no puede esquivarlos y por tanto a veces se producen situaciones en que el bot es devorado al encontrarse un fantasma en su recorrido, si es que hay píldoras por el camino, de ahí que muchas veces no mejore el tiempo que sobrevive respecto al de un bot aleatorio.

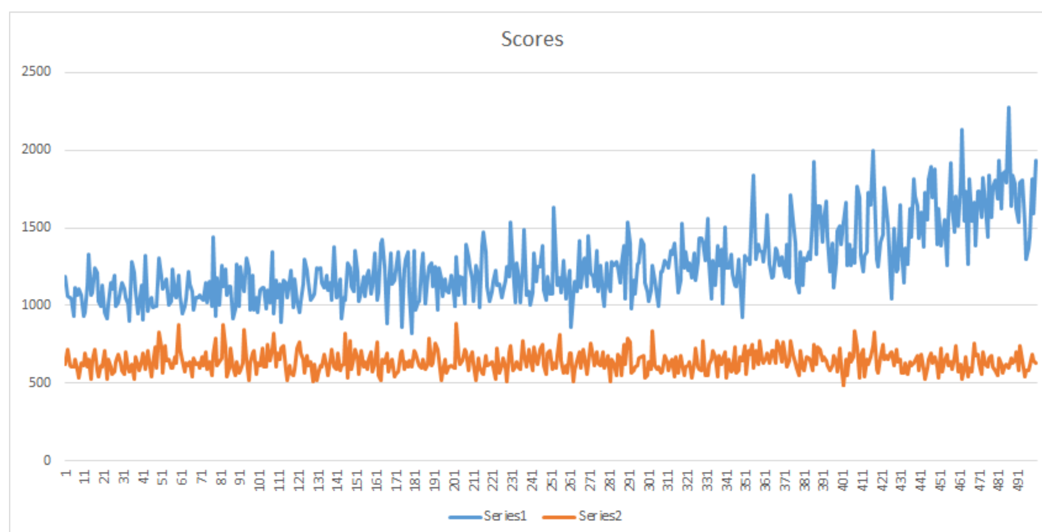

Figura 5.1 Puntuaciones del experimento 1.

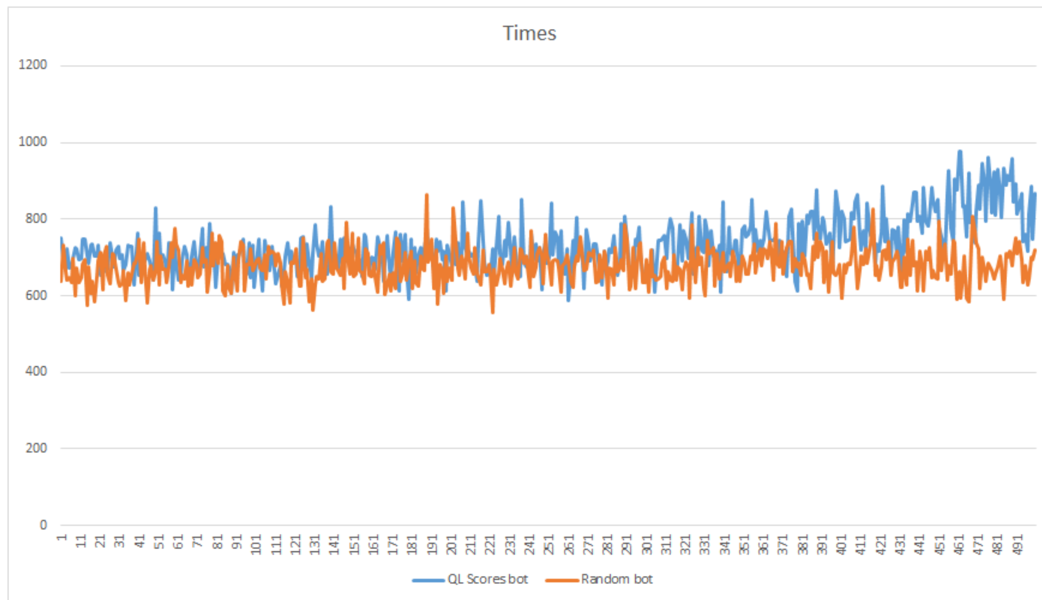

Figura 5.2 Tiempos del experimento 1.

5.1.2. Experimento 2

Para el segundo experimento, el objetivo es que Pac-Man aprenda otro tipo de comportamiento. En este caso el agente únicamente obtiene del juego información sobre los fantasmas, y la recompensa de cada acción tomada dependerá del tiempo que siga vivo en el laberinto. Con esto se pretende reforzar las acciones que eviten que el bot sea devorado por los fantasmas, y así conseguir que aprenda a huir de ellos.

Resultados: En este caso el bot ha aprendido a escapar de los fantasmas, dando media vuelta cuando ve que están cerca, cosa que se puede observar claramente mirando el tiempo que sobrevive en las gráficas, mucho mayor que en el experimento anterior.

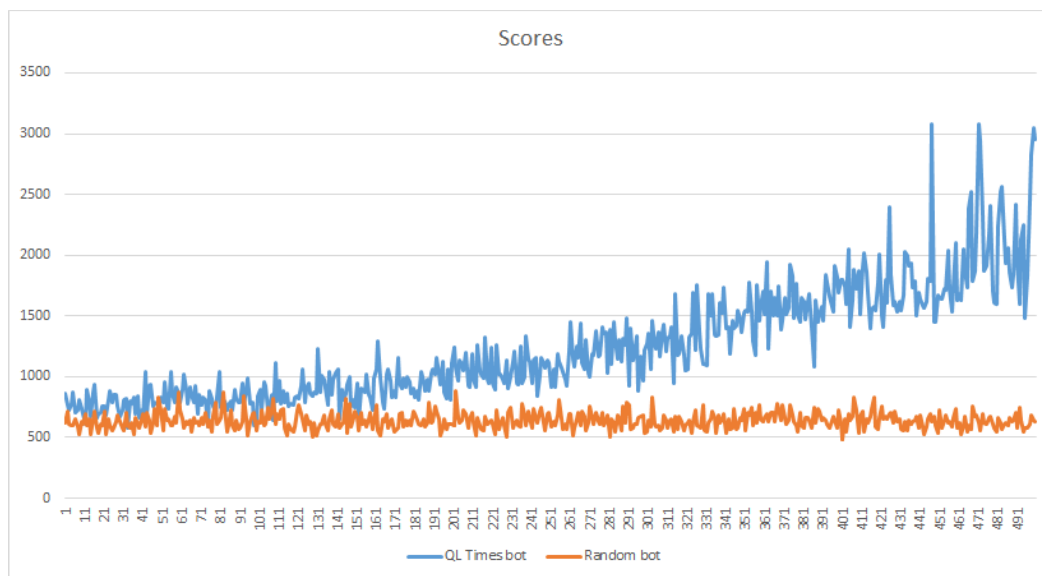

Figura 5.3 Puntuaciones del experimento 2.

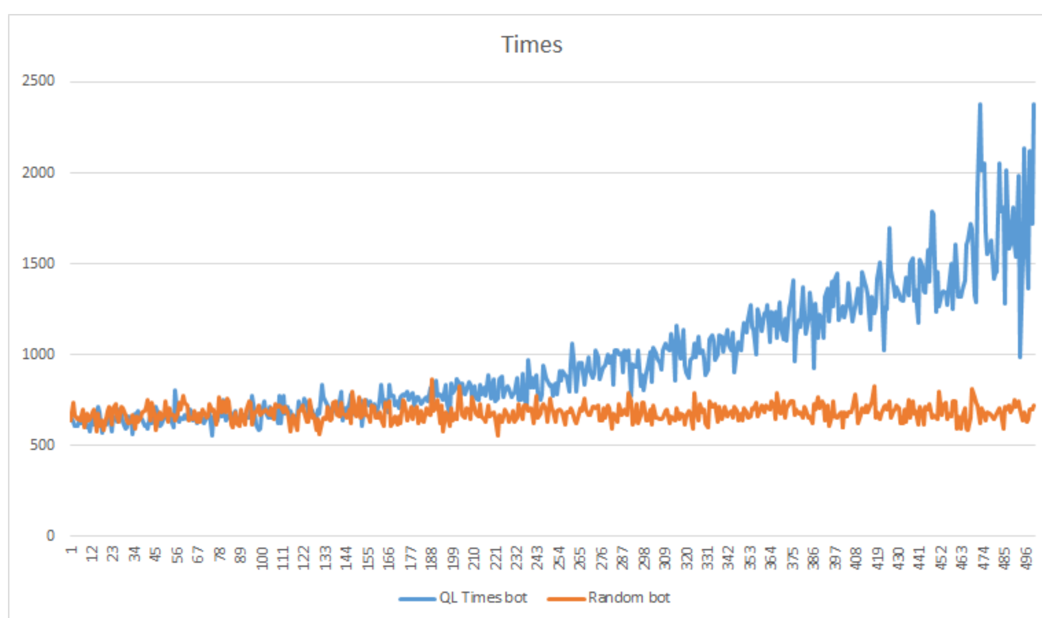

Figura 5.4 Tiempos del experimento 2.

5.2. Experimentos y resultados de RL/CBR

En esta sección intentaremos mejorar las técnicas vistas de aprendizaje por refuerzo, mezclando el algoritmo Q-learning con el razonamiento basado en casos.

Ahora es cuando se utilizará todo lo visto sobre CBR: manejo de la base de casos, comparar casos utilizando funciones de similitud, etc., y compararemos estos experimentos con los que se realizaron en la sección anterior, además de con el bot aleatorio también mencionado previamente.

5.2.1. Experimento 1

En este primer experimento, el bot recibe información sobre las píldoras, píldoras especiales, y sobre los fantasmas cuando están en estado comestible, y establecemos que la recompensa que recibe por cada acción esté basada en la puntuación. El objetivo de esto es que el agente aprenda qué acciones van a reportarle mayor puntuación en cada momento.

El vector de pesos utilizado ha sido $w = (0.9, 0, 0.05, 0.05, 0)$, dando total prioridad a las píldoras por encima de las píldoras especiales y los fantasmas comestibles.

Resultados: El agente aprende a perseguir las píldoras, que son las que le reportan mayor beneficio. Elige caminos que contienen la mayor cantidad de ellas, desviándose a veces si ve que tiene un fantasma comestible cerca.

Se observa en la gráfica que mejora mucho las puntuaciones obtenidas, respecto a los bots de los experimentos anteriores, que empleaban solo Q-learning, tanto al que se centraba en obtener puntos como al que tenía como objetivo sobrevivir la máxima cantidad de tiempo.

A pesar de que el bot no conoce la localización de los fantasmas, las puntuaciones obtenidas son bastante aceptables, llegando a conseguir más de 7000 puntos en algunos episodios.

En las figuras 5.5 y 5.6 se pueden ver las gráficas de los resultados de este experimento.

Se puede ver un vídeo de demostración de este experimento a través del siguiente enlace: https://youtu.be/-fD8t7tRjG0

5.2.2. Experimento 2

Para el segundo experimento, se intentará que Pac-Man aprenda otro tipo de comportamiento. En este caso el agente obtiene del juego información sobre los fantasmas, y la recompensa de cada acción tomada dependerá del tiempo que siga vivo en el laberinto. Con esto se pretende reforzar las acciones que eviten que el bot sea devorado por los fantasmas, y así conseguir que aprenda a huir de ellos.

El vector de pesos utilizado ha sido $w = (0, 0.95, 0, 0.05, 0)$, dando total prioridad a lOs fantasmas por encima de los fantasmas comestibles.

Resultados: Como se esperaba, el agente aprende a esquivar a los fantasmas, tomando rutas alternativas y así sobrevivir cada vez más tiempo en el laberinto. Se observa en la

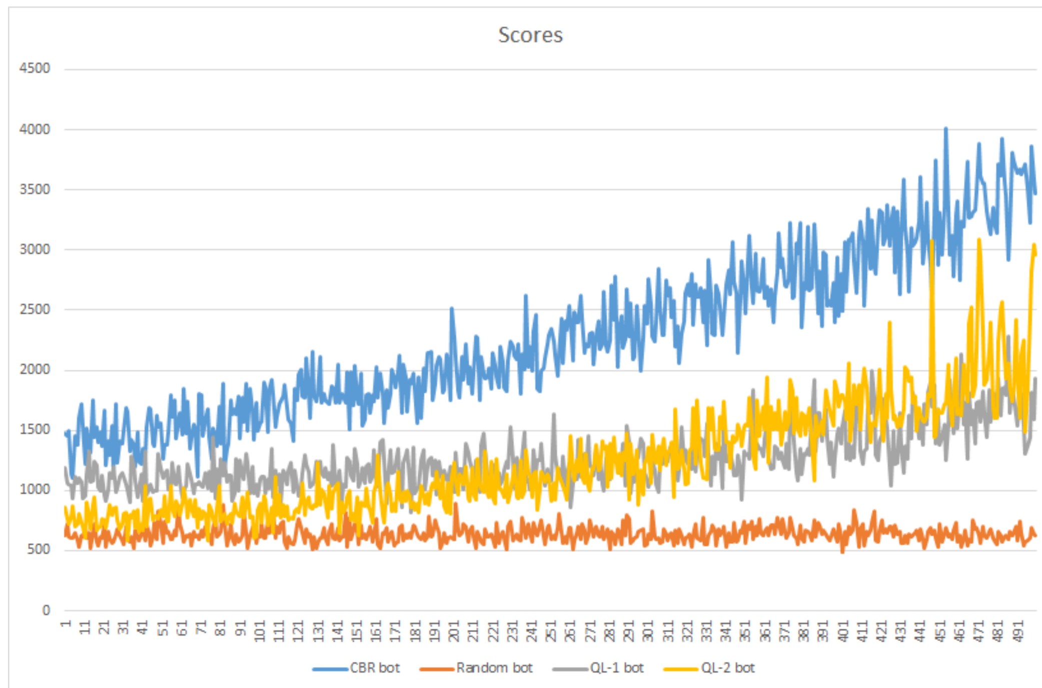

Figura 5.5 Puntuaciones del experimento 1.

gráfica 5.8 que mejora los tiempos del bot que utilizaba únicamente Q-learning para aprender a huir de los fantasmas, consiguiendo tiempos máximos mucho mayores.

Se puede ver un vídeo de demostración de este experimento a través del siguiente enlace: https://youtu.be/4phEjPAGrbk

5.2.3. Experimento 3

El tercer y último experimento tratará de combinar los dos anteriores, para comprobar si el agente puede utilizar ambos comportamientos aprendidos (comer píldoras y esquivar fantasmas) para pasarse niveles o, al menos, jugar de forma inteligente. Las recompensas dependerán ahora tanto de la puntuación como del tiempo que sobreviva el bot, pero cada una será ponderada mediante los pesos de la función de similitud, para reforzar acciones que involucran a los parámetros a los que el agente le da más prioridad.

El vector de pesos utilizado ha sido $w = (0.2, 0.7, 0.05, 0.05, 0)$, dando más prioridad a las píldoras por encima de lo demás.

Resultados: El objetivo era que el bot aprendiera ambas estrategias al mismo tiempo, de manera que pudiera aprender también cuándo utilizar una y cuándo otra, para poder así superar niveles con mayor solvencia. El resultado de este experimento parece quedarse a medio camino de lo que se pretendía conseguir, viendo que el bot toma decisiones extrañas de

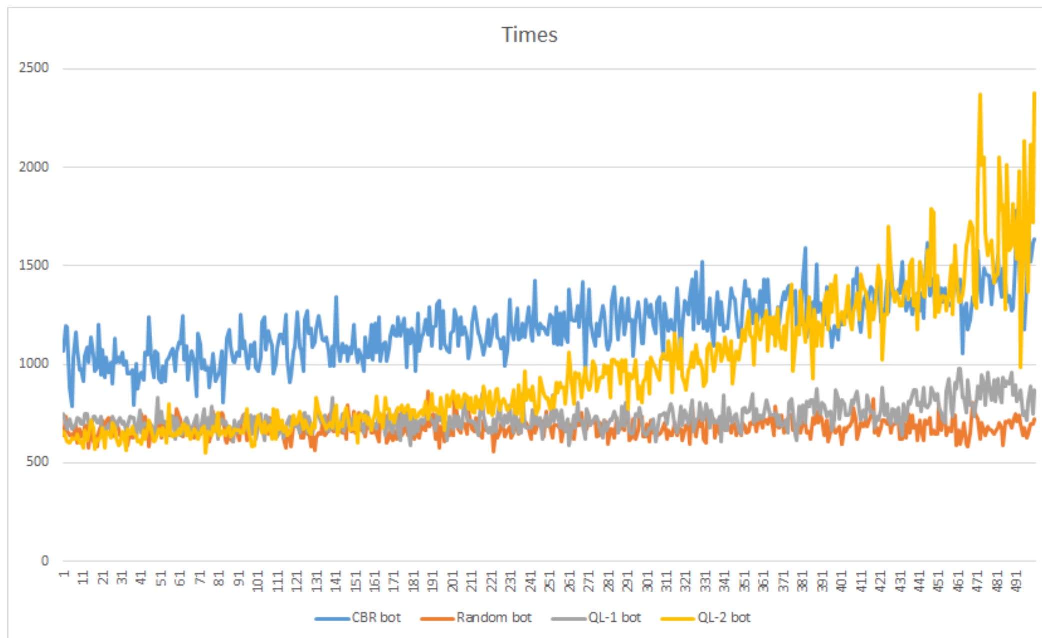

Figura 5.6 Tiempos del experimento 1.

vez en cuando, o se queda atascado entre dos casos que le dan instrucciones contradictorias en situaciones muy específicas. Sin embargo, supera los 4000 o 5000 puntos en un buen número de episodios, con lo que parece un buen punto de partida para empezar a mejorarlo.

En las gráficas 5.9 y 5.10 se puede ver la progresión tanto de las puntuaciones como de los tiempos obtenidos.

Se puede ver un vídeo de demostración de este experimento a través del siguiente enlace: https://youtu.be/fftrYjzTLPg?list=PL1yc2hRaiSiDkHGixdYWxN7yN_xEHGmuX

A continuación se muestra una tabla con algunas estadísticas recogidas en los experimentos, que muestran el rendimiento que han tenido los bots construidos.

En ella se recogen datos como la puntuación y tiempos máximos y medios obtenidos en la simulación de cien partidas adicionales, en las que el bot no sigue aprendiendo, con los conocimientos adquiridos en cada uno de los experimentos. También se muestra información acerca de los niveles superados, en concreto el máximo número de niveles que ha superado cada bot.

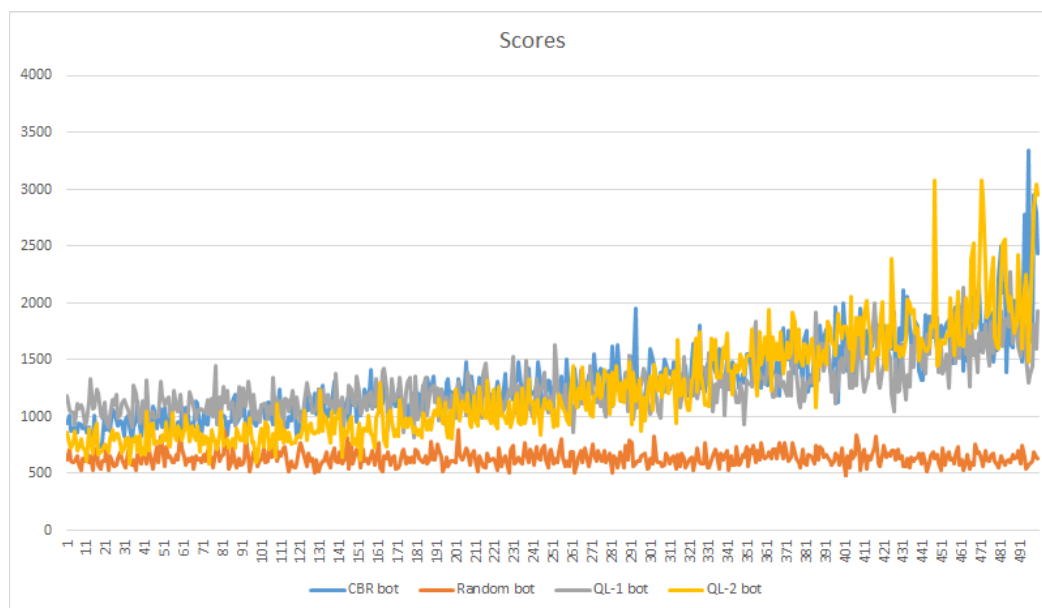

Figura 5.7 Puntuaciones del experimento 2.

Bot CBR	Max Puntuación	Max Tiempo	Max Niveles	Med Puntuación	Med Tiempo
Experimento 1	8880	4930	2	3438	1193
Experimento 2	11690	8976	3	2364	2478
Experimento 3	6710	4666	2	1950	1461

Cuadro 5.1 Estadísticas de los experimentos con CBR

Como se puede observar en la tabla, el primer bot obtiene una puntuación media mucho mayor que los demás, aunque el segundo, al aprender a sobrevivir más tiempo, al final obtiene una puntuación máxima más grande, y también supera más niveles. Juntar ambas estrategias es un reto mayor, como se puede observar en las puntuaciones y tiempos medios del tercer bot. Sin embargo, en sus mejores episodios obtiene unas marcas de tiempo y puntos bastante aceptables, lo que es esperanzador para seguir investigando este tipo de métodos.

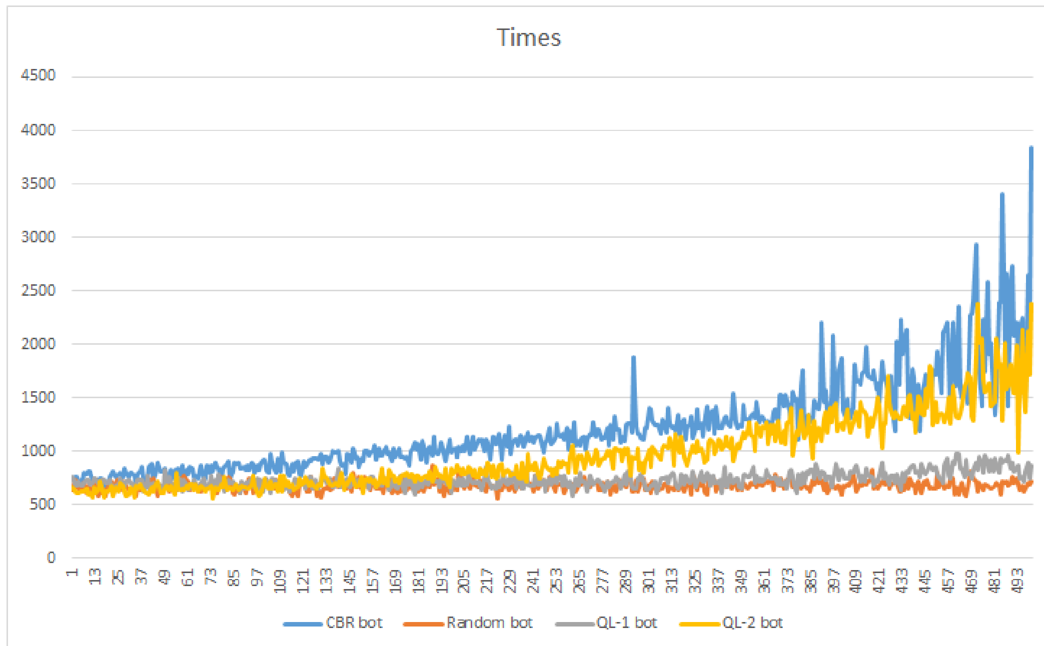

Figura 5.8 Tiempos del experimento 2.

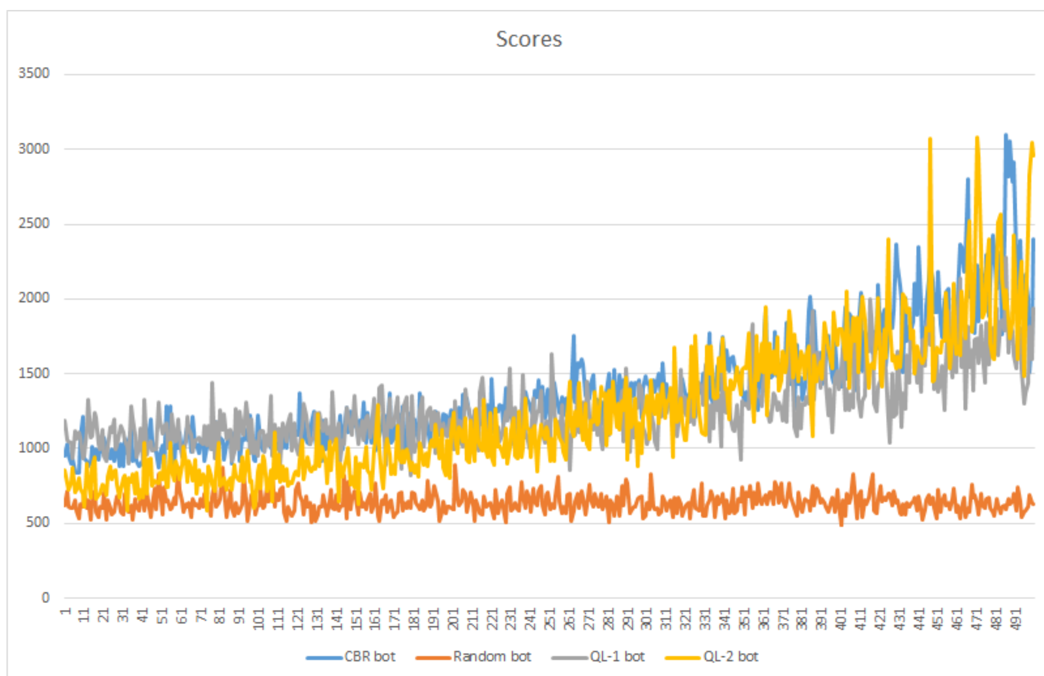

Figura 5.9 Puntuaciones del experimento 3.

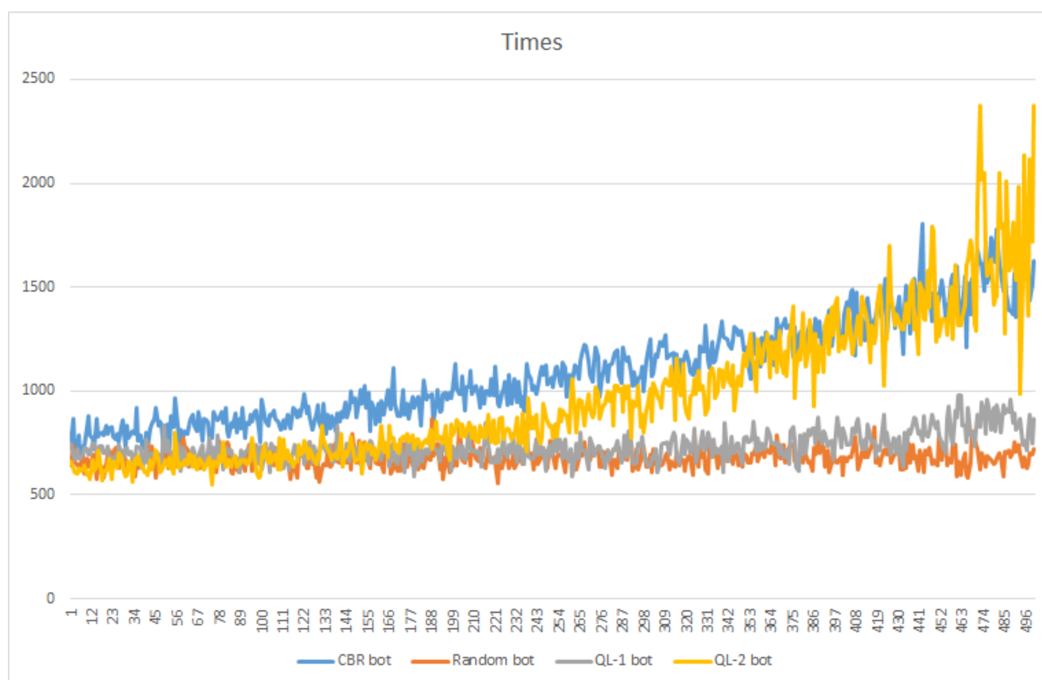

Figura 5.10 Tiempos del experimento 3.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Uno de los mayores retos de este proyecto ha sido la implementación de todo el código y el diseño de todo lo necesario para operar con el videojuego, ya que se disponía de muy poco trabajo previo en el que basarse. Por otro lado, esto pasó a ser una motivación más, la de conseguir sentar unas bases en la aplicación de RL y CBR de manera conjunta a los videojuegos en general, y a Pac-Man en particular.

Las conclusiones extraídas de todo este proceso podrían resumirse como las siguientes:

- El razonamiento basado en casos es sin duda un método que parece efectivo para este tipo de aplicaciones. Sin embargo, requiere de un gran trabajo de diseño, y de realizar muchos experimentos para estudiar la viabilidad de las soluciones propuestas. En concreto lo que más problemas ha dado ha sido la tarea de diseñar los casos, y de cómo utilizar la información que contienen para hacer que el sistema evolucione a medida que se simulan las partidas. Conseguir un equilibrio para que los casos no sean demasiado específicos (y por tanto se reutilicen pocas veces), ni tampoco demasiado generales (y no aporten soluciones especialmente útiles), es complicado y requiere de muchas pruebas hasta encontrar los parámetros que mejor se adaptasen al objetivo perseguido.

- El videojuego que se utiliza finalmente, PacMan, parece a priori sencillo pero representa un dominio que se puede volver extremadamente complejo cuando pretendes que un bot aprenda a jugar. Un jugador humano procesa información visual y la aplica muy rápido cuando toma decisiones, pero para conseguir que una máquina haga lo mismo, se necesita un grado de abstracción de esa información que en ocasiones se vuelve muy difícil de conseguir.

- En cuanto a progresar en el juego, se ha comprobado que es viable que el bot aprenda estrategias determinadas, pero mezclarlas todas juntas para que se pase niveles es un problema mucho más grande, que debe ser estudiado con mayor profundidad.

- El tiempo ha sido una traba en el desarrollo del proyecto, dado que ha habido que tener siempre un ojo en el rendimiento del software para que las simulaciones no se hicieran demasiado largas. En mi caso, al empezar a implementar CBR sobre lo que ya tenía de RL, me encontraba con que la simulación de unos pocos de experimentos podía llevar varias horas, por lo que era realmente necesaria una solución a esto. Finalmente después de revisar mucho el código, se consiguó bajar esos tiempos hasta conseguir que esas mismas simulaciones se realizaran en unos pocos minutos. Acelerar estos procesos ha sido vital para poder obtener mejores resultados, pero al mismo tiempo ha sido difícil desarrollar este código para que trabaje de forma eficiente. Todo esto ha ralentizado el proceso de desarrollo en algunos momentos, lo que quizá haya sido un lastre a la hora de llegar a mejores resultados.

6.2. Trabajo futuro

En el campo de los videojuegos y la inteligencia artificial queda un mundo por explorar. Como se ha mencionado anteriormente, la información de que se disponía para realizar este trabajo era más bien escasa, así que el trabajo realizado puede considerarse una primera piedra en la investigación de CBR aplicado a PacMan.

Para proseguir con esta investigación, algunas de las tareas que se podrían continuar son:

- Estudiar distintas representaciones de un caso. Si se consiguen mejores abstracciones de la información del juego, quizá se puedan llegar a mejores resultados. Un primer paso podría ser obtener una discretización de las distancias más efectiva, que cubra un mayor rango de posibilidades y por tanto dote de un conocimiento más útil al bot.

- Investigar qué otras funciones de similitud pueden ayudar al algoritmo a aprender una base de casos efectiva. Con nuestra representación de un caso sería interesante aprender cómo conseguir una distribución de parámetros óptima, con el fin de que el bot consiga entrelazar estrategias y aprender cuándo debe usar una y cuándo otra.

- Mejorar las tareas de mantenimiento de la base de casos. Estudiar mejores métodos para el borrado de casos, y para el indexado y búsqueda de los mismos.

- Explorar otros dominios. Implementar CBR con otros videojuegos para seguir descubriendo los límites de este tipo de algoritmos.

- Investigar otros métodos de aprendizaje o razonamiento aplicados a Pac-Man, para comparar los resultados. Un ejemplo sería investigar cómo podrían aplicarse a este juego métodos como redes neuronales o algoritmos genéticos, y comprobar si son más o menos adecuados.

6.3. Difusión

A partir de lo investigado durante la realización de este proyecto, se ha escrito un artículo para el IV Congreso de la Sociedad Española para las Ciencias del Videojuego (CoSECiVi 2017), actualmente en estado de revisión.

Si se deseara continuar con esta investigación, se recuerda que el código completo de este proyecto puede encontrarse en https://github.com/fedomi/PacMan-vs-Ghosts.git

El código se encuentra bajo la licencia libre GPLv2.0: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html

La documentación se publica bajo una licencia CC BY-NC-SA.

Bibliografía

[1] Ms. pac-man competition (2007-2011). http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html

[2] Ms. pac-man vs ghosts ai. http://www.pacmanvghosts.co.uk

[3] Aamodt, A., Plaza, E.: Case-based reasoning: Foundational issues, methodological variations, and system approaches. AI Commun. 7(1), 39–59 (Mar 1994)

[4] Bom, L., Henken, R., Wiering, M.: Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In: ADPRL. pp. 156–163. IEEE (2013)

[5] Gallagher, M., Ledwich, M.: Evolving pac-man players: Can we learn from raw input? In: Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007, Honolulu, Hawaii, USA, 1-5 April, 2007. pp. 282–287 (2007)

[6] Lample, G., Chaplot, D.S.: Playing FPS games with deep reinforcement learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 2140–2146 (2017)

[7] Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA (1998)

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (02 2015)

[9] Pyeatt, L.D., Howe, A.E.: Decision tree function approximation in reinforcement learning. Tech. rep., In Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models (1998)

[10] Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA, 1st edn. (1998)

[11] Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, King's College, Cambridge, UK (May 1989)

[12] Williams, P.R., Liebana, D.P., Lucas, S.M.: Ms. pac-man versus ghost team CIG 2016 competition. In: IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016. pp. 1–8 (2016)