Aprendizaje Automático
Práctica 3

García Fernández, Antonio 100346053
García García, Alba María 100346091

Leganés G83

15 Mayo 2018

Índice

1. Introducción

En esta práctica se propone la creación de un agente automático mediante el uso de la técnica de aprendizaje *Qlearning*. Para ello, se nos proporcionó a los estudiantes una implementación de dicho algoritmo que éramos libres de modificar para ajustarlo al problema propuesto.

La práctica se divide en la descripción de todas las decisiones y presunciones llevadas a cabo por los estudiantes; una explicación del funcionamiento del código implementado y unos comentarios y conclusiones acerca de el comportamiento del agente implementado y los resultados obtenidos.

2. Descripción del espacio de estados

A diferencia que el resto de prácticas que hemos realizado hasta la fecha en esta asignatura, **no vamos a necesitar balancear las instancias en este caso**. Esto se debe a que el propósito de este método de aprendizaje es obtener instancias, clasificarlas en uno de los estados posibles y obtener su refuerzo respecto a una ventana temporal elegida, ya sea esta *n+6*, *n+12* o *n+24* ticks. De acuerdo al estado en que se encuentra Mario, la acción que realizó y el refuerzo que obtuvo; iremos actualizando la tabla Q que luego servirá para definir la política del agente automático. Por tanto, **recibir más instancias de un estado que de otro** no es un problema, **sólo causará que la tabla Q obtenga resultados más precisos para ese estado con mayor antelación que con los otros**.

Debido a que los **dos objetivos de Mario** en esta práctica son maximizar tanto la distancia recorrida como la *intermediate reward*, hemos considerado aquellos elementos clave del mapa que nos permiten alcanzar (o no) dichos objetivos para la configuración de nuestro espacio de estados.

Si queremos **maximizar la distancia recorrida**, debemos tener en cuenta:

- Los posibles **enemigos** que nos podemos encontrar en el camino pueden herir a Mario y hacer que no se pase el nivel.
- Los **obstáculos** imposibilitan seguir avanzando a Mario, aunque no representan un peligro directo a la hora de perder vida.

Por otro lado, si queremos **maximizar la *intermediate reward*** obtenida, debemos tener en cuenta:

- Los **enemigos** en esta situación juegan un doble papel: si hieren a Mario, bajará la recompensa obtenida; mientras que si es Mario quien acaba con ellos, esta subirá.
- Tanto las **monedas** como los **champiñones** aumentan la *intermediate reward* de Mario, siempre que vaya a recogerlos. Al igual que en la Práctica 2, no consideraremos ni flores de fuego ni bloques ?; ya que en el caso del primero, este no se encuentra nunca en el área inmediata a Mario (no baja hasta el suelo una vez que sale en pantalla tras golpear el bloque en el que se encontraba, como sucede con los champiñones) y el segundo no ofrece dicha recompensa de forma asegurada.

Sin embargo, los elementos descritos anteriormente pueden aparecer simúltaneamente en una misma situación dentro del entorno del juego; por lo que a la hora de asignar a qué estado pertenece una instancia, tendremos que definir una **serie de prioridades**:

- Consideramos que aquello que **Mario debe evitar a toda costa** es **perder vida** ya que afecta a la consecución de los dos objetivos, por tanto, la primera prioridad es detectar que hay enemigos cercanos.
- Tras esto, hay que tener en cuenta que el objetivo de aumentar la *intermediate reward* es mucho más complicado que avanzar lo máximo posible. Por tanto, **la segunda prioridad será la de recoger monedas o champiñones** que se encuentren cercanos al entorno de Mario.

- Finalmente, sólo nos quedaría **evitar los obstáculos** para que Mario avance tanto como pueda en el nivel.

Tras este análisis, ya podríamos definir el espacio de estados:

- En cuanto a los **enemigos**, **los que más nos interesan son aquellos que tenemos a la derecha**, ya que serán los que impidan avanzar a Mario. En este caso es muy importante saber si el agente se encuentra en el suelo o en el aire, ya que las acciones que puede realizar son distintas en cada caso. Así, si se encuentra en el suelo, el comportamiento óptimo sería saltarlo ya sea para acabar con él o evitarlo. Para estos casos necesitamos saber también si Mario saltó en el tick anterior, ya que el mecanismo del salto no funciona si el agente se encuentra en el suelo, saltó en el tick anterior e intenta saltar de nuevo. Sin embargo, si se encuentra en el aire, lo mejor que puede hacer Mario es moverse para atrás y evitarlo, ya que no puede saltar mientras esté en esta situación. **También nos podemos encontrar enemigos a la izquierda y justo encima**; aunque no sea tan frecuente como encontrarlos delante. No creemos que sea necesario diferenciar aquí si Mario está pisando el suelo o no, ya que lo único útil que puede hacer si se encuentra un enemigo encima es evitarlo, ya sea moviéndose a la derecha o a la izquierda. De igual modo, en el caso de los enemigos que están detrás del agente, no merece la pena ir a por ellos para intentar matarlos, ya que no obstaculizan a Mario y puede ser un riesgo para él. Intentaremos en estos casos que Mario aprenda a evitarlos, por lo que no nos interesa saber si el agente se encuentra o no tocando el suelo. Finalmente, **que Mario se encuentre con un enemigo justo debajo es irrelevante**, ya que lo pisará irremediablemente sin tener que hacer nada. Si no lo consigue, esta situación pasará a la de tener un enemigo delante o detrás. Por tanto, las situaciones serían las siguientes:

  - **Mario tiene un enemigo arriba**.
  - **Mario está en el suelo, saltó en el tick anterior y tiene un enemigo a la derecha**.
  - **Mario está en el suelo, no saltó en el tick anterior y tiene un enemigo a la derecha**.
  - **Mario está en el aire y tiene un enemigo a la derecha**.
  - **Mario tiene un enemigo a la izquierda**.

  Dentro de esta categoría podemos encontrar **situaciones en las que Mario tiene enemigos en distintas posiciones**. Por tanto, hemos establecido prioridades dentro de esta categoría también. Como nos parece bastante complicado acabar con los enemigos, **vamos a priorizar que nuestro agente los evite**. Tras ver cómo se comportaban todos los bot que hemos creado hasta la fecha, hemos comprobado que la posición donde más difícil le resulta evitarlos en arriba; así que comprobaremos si Mario tiene enemigos arriba primero. Tras esto, los enemigos a la izquierda no suelen suponer ningún peligro a menos que el agente esté atascado, por lo que después miraremos si tiene algún enemigo a la derecha. Finalmente, **debido al poco interés por acabar con los enemigos a la izquierda de Mario, colocaremos esta situación como la penúltima de mayor prioridad**. Esto nos permite también avanzar en el caso de que el agente se encuentre bloqueado y con algún enemigo a su izquierda. Estas situaciones son peligrosas si lo único que está haciendo el agente es moverse a la derecha sin saltar el obstáculo, lo que acabaría hiriendo a Mario.

- Una vez terminados los enemigos, debemos centrarnos en **las monedas y los champiñones**. La gran diferencia con el caso anterior es que ninguno de estos objetos pueden herir a Mario; así que lo que nos interesa es saber **en qué posición se encuentran**, para que Mario pueda ir a recogerlos. Consecuentemente, no será necesario saber si Mario se encuentra o no en el aire, ya que no son objetos que deba evitar. Así, al igual que en caso anterior, consideraremos las **posiciones delante, detrás y encima de Mario**. Aquí tampoco será necesario saber si existe alguna moneda o champiñón justo debajo, ya que si el agente no lo recoge al caer, se

encontrará a su derecha o izquierda tras tocar el suelo, casos que ya hemos considerado. De igual modo, no es necesario hacer ningún tipo de distinción entre monedas y champiñones, ya que ambos tienen el mismo comportamiento: aumentar la *intermediate reward*. Sin embargo, y al igual que sucede con las situaciones de los enemigos, **sí que es necesario comprobar si Mario saltó en el tick anterior para el estado en que las monedas se encuentran arriba**; ya que lo recomendable aquí es que el agente salte a por las monedas o champiñones. Incluiremos por tanto esta condición en todas las situaciones en las que sea vital saltar para que Mario no se quede atascado o pierda vida innecesariamente. Con todo esto, la situaciones serían:

- **Mario tiene una moneda o champiñón a la izquierda**.
- **Mario tiene una moneda o champiñón arriba y saltó en el tick anterior**.
- **Mario tiene una moneda o champiñón arriba y no saltó en el tick anterior**.
- **Mario tiene una moneda o champiñón a la derecha**.

Al igual que sucedía con los enemigos, **Mario se puede encontrar con monedas o champiñones en distintas posiciones de su entorno inmediato**; lo que nos obliga a ordenar las prioridades de estos estados otra vez. Este orden esta basado en que, como consecuencia de que Mario se mueve normalmente a la derecha para maximizar la distancia recorrida, **el único estímulo que tiene para moverse en otra dirección son aquellos elementos que le proporcionan *intermediate reward***. Con este razonamiento nos queda claro que el último área en el que Mario se debe centrar es el de su derecha. De este modo, nos quedarían las zonas de la izquierda y arriba. El problema de que el agente coja las monedas o champiñones que se encuentren arriba de él con máxima prioridad es que avance hacia delante mientras salte. Esto haría que perdiera aquellas monedas o champiñones que se encuentren detrás, por lo que esta área debe ser nuestra máxima prioridad.

- Tras esto, debemos considerar aquellos casos en los que **Mario está bloqueado por algún tipo de obstáculo**. En este caso en concreto y debido a que para maximizar la distancia recorrida Mario se debe mover hacia la derecha, **sólo creemos necesario considerar aquellos casos en los que haya un obstáculo justo a la derecha de Mario**. Otra opción sería considerar también aquellas situaciones en las que Mario no sólo se encuentra bloqueado por la derecha, sino por arriba también; ya que ahí saltar hacia delante no resuelve nada (que sería el movimiento más acertado si el obstáculo estuviera sólo a la derecha). Sin embargo, hemos tenido que desechar este estado, ya que requeriría una estrategia que consistiera en moverse hacia atrás lo suficiente como para que al saltar se llegue al obstáculo que se encontraba anteriormente encima de Mario. Sin embargo, es bastante complicado saber cuánto se tiene que mover para atrás y cuándo exactamente se pertenece a ese estado. Sabiendo en qué posición tenemos que mirar para saber si Mario está bloqueado, **tenemos que conocer ahora si el agente se encuentra en el suelo o no, al igual que si saltó en el tick anterior**. Esto es **muy importante para el mecanismo del salto**, ya que Mario no puede saltar en dos ticks seguidos si en el segundo se encuentra en el suelo. Sólo de este modo podremos conseguir que Mario no se quede atascado al estar todo el rato manteniendo pulsado el botón de salto al intentar saltar algún obstáculo. Por tanto, obtendríamos los siguientes estados:

  - **Mario está en el suelo, no saltó en el tick anterior y tiene un obstáculo a la derecha**.
  - **Mario está en el suelo, saltó en el tick anterior y tiene un obstáculo a la derecha**.
  - **Mario está en el aire y tiene un obstáculo a la derecha**.

  En este caso no necesitamos aplicar ningún tipo de prioridad, ya que los tres estados son complementarios y no pueden suceder al mismo tiempo.

- Finalmente, la única situación que nos queda es aquella en la que **no aparecen ninguno de los posibles elementos clave del mapa que consideramos**. Por tanto, no habría ningún tipo de enemigo cerca, al igual que tampoco monedas, champiñones u obstáculos. En estos casos, lo único que debería hacer Mario sería seguir avanzando, ya que no podría aumentar su *intermediate reward*. Así:
  - **Mario no tiene ningún enemigo, moneda, champiñón u obstáculo cerca**.

Con estos trece estados quedaría definido nuestro espacio de estados. Así, la pertenencia o no a estos estados se explica a través de los atributos que usemos para clasificar las instancias. Por tanto, las cuestiones que debemos aclarar ahora son: cómo va a ser entrenado el agente; qué acciones va a poder realizar; y cuál es el rango en celdas que abarcamos para saber dónde se encuentran tanto enemigos, como monedas, champiñones y obstáculos. Sin embargo, estas cuestiones forman parte del siguiente apartado.

3. Descripción de los atributos seleccionados

Los atributos que elijamos para recoger instancias deben servir tanto para saber en qué estado se encuentra el agente como cuál es el refuerzo que obtiene al ejecutar una determinada acción dentro de dicho estado. Por tanto, antes de empezar a hablar sobre los atributos que hemos elegido, necesitamos saber **cómo será el agente que realizará la recogida de instancias de entrenamiento**.

- **Cuestión 1: ¿El agente debe ser un bot o estar controlado por un humano?**

  Aunque **al principio pensamos que la mejor opción para entrenar a nuestro agente pasaba por usar sólo el humano**, nos dimos cuenta de que esto no era así una vez que empezamos a ponerlo en práctica. Esto se debe principalmente a que el tiempo de reacción del humano es bastante bajo y comete muchos errores; por lo que es difícil entrenar al agente en aquello que está bien. Otro asunto importante es que algunas situaciones requieren de un conjunto de acciones bastante específico que es muy complicado de ejecutar si no eres un bot. Así que, tras muchos intentos y fracasos, **probamos a ver cómo aprendía Mario haciendo uso del agente procedural que implementamos en el Tutorial 1**. Vimos entonces que grabando las instancias que obtenía tras recorrerse sólo el nivel de la semilla 0 para la dificultad más baja, **el agente automático de esta práctica emulaba a la perfección sus movimientos**. En cuanto a la tabla Q, a pesar de que no se habían inicializado todos sus valores, aquellos que estaban presenten tenían mucho sentido.

  Tras ver esto, nos planteamos **implementar un comportamiento más complejo para el bot que recoge las instancias**, de modo que tuviera en cuenta todos los estados que hemos descrito a la hora de ejecutar las acciones. Haciendo uso de esta idea, hemos visto que nuestro agente ha aprendido cuál es la acción o acciones más adecuadas para cada una de las situaciones planteadas. Por tanto, **para poder obtener una tabla Q completa sólo tenemos que hacer uso del humano**. Este agente tiene la responsabilidad de recoger las instancias de aquellas combinaciones de situaciones y acciones que el bot no contempla. De este modo, hemos adoptado un **método de recogida de instancias mixto**.

- **Cuestión 2: ¿Qué acciones podrá realizar el agente automático creado a través de aprendizaje por refuerzo?**

  A parte de las acciones que se indican en el enunciado de la práctica (**avanzar, retroceder, saltar y avanzar saltando**); vamos a considerar también **retroceder saltando y quedarse quieto**. En cuanto a la primera, la incluimos porque era la análoga de avanzar saltando y así no se perdía el abanico completo de movimientos. Luego, con respecto a la segunda, hemos considerado que puede ser necesaria en situaciones de corta duración que requieran que Mario

no avance pero que tampoco retroceda. Finalmente, una última acción que nos planteamos introducir mientras implementábamos el bot encargado de recoger algunas instancias fue la de **avanzar saltando durante dos ticks**. Esta acción es totalmente necesaria para evitar que **Mario se quede bloqueado debido a la prioridad que existe entre los estados definidos**. Por ejemplo, si hay monedas encima de un obstáculo, Mario se centrará en que tiene monedas a su derecha e intentará acercarse a ellas sin saltar de forma indefinida. Se trata de una acción durativa porque comprobamos que saltando hacia la derecha durante sólo un tick le seguía dejando bloqueado en la mayoría de las situaciones. Así, las acciones tomarían los siguientes valores:

- **Avanzar** $\longrightarrow$ ***right***.
- **Retroceder** $\longrightarrow$ ***left***.
- **Saltar** $\longrightarrow$ ***jump***.
- **Avanzar saltando** $\longrightarrow$ ***right_jump***.
- **Avanzar saltando durante dos ticks** $\longrightarrow$ ***right_jump_long***.
- **Retroceder saltando** $\longrightarrow$ ***left_jump***.
- **Quedarse quieto** $\longrightarrow$ ***still***.

Una vez aclaradas estas cuestiones, ya podemos definir los **atributos que necesitamos para saber en qué estado se encuentra una instancia**. Dichos atributos serían los siguientes:

- ***isOnGround***. Este atributo puede tomar dos valores: *true* si Mario se encuentra pisando el suelo en ese tick y *false* en el caso contrario. Nos será necesario para determinar esos estados que requiere que Mario se encuentre en el suelo o en el aire.
- ***isEnemy***. En este caso no es suficiente con *true* o *false*, ya que los estados requiren saber en qué posición se encuentra el enemigo. Por ello, este atributo tomará los valores: *right*, *left*, *up* y *none*; dependiendo si el enemigo se encuentra a la derecha, izquierda, arriba o si directamente no hay enemigos cerca. Al igual que sucedió en los estados que incluían enemigos, hemos establecido aquí el mismo orden de prioridades para determinar en qué posición se encuentra dicho enemigo, esto es: primero *up*, luego *right* y por último *left*. Explicaremos cuál es el rango que toma este y los otros atributos una vez que expliquemos en qué consisten el resto.
- ***isCoinMushroom***. Al igual que *isEnemy*, tomará los valores *right*, *left*, *up* y *none*; con el mismo significado que antes. Esto se debe a que los estados que involucran monedas y champiñones también necesitan saber la posición de los mismos. Igualmente, tendremos que establecer prioridades; que serán las mismas que las establecidas por las situaciones que se basaban en detectar monedas o champiñones cercanos: primero *left*, después *up* y por último, *right*.
- ***isBlocked***. Con este atributo volvemos a los valores binarios (*true* y *false*), ya que sólo necesitamos evaluar si el agente se encuentra bloqueado en una sola dirección (justo a su derecha).
- ***jumped***. Finalmente, haciendo uso de *jumped* podremos saber si Mario saltó en el tick anterior (*true*) o no (*false*). Se trata por tanto del atributo clave para el mecanismo del salto, necesario tanto para saltar enemigos como obstáculos.

Una vez que conocemos los atributos que necesitamos, podemos tomar la decisión de **guardar el estado al que pertenece la instancia como un atributo más** o calcularlo antes de pasarlo por la tabla Q. Siempre hemos pensado (como se pudo ver en la Práctica 2) que cuanto antes tengamos la información disponible mucho mejor, así que hemos decidido guardarla como un atributo más. Siguiendo esta línea de pensamiento, decidimos **incorporar como atributo el estado que se da dentro de 12 ticks** para simplificar nuestra tarea en el bot destinado a aplicar *QLearning*. Por tanto, y siguiendo el orden de prioridades que elegimos para los estados, estos atributos, ***state*** y ***state+12***, puede tomar los siguientes valores:

- ***enemy_up***: necesita *isEnemy = up*.
- ***enemy_right_ground_jumped***: necesita *isOnGround = true*, *isEnemy = right* y *jumped = true*.
- ***enemy_right_ground***: necesita *isOnGround = true*, *isEnemy = right* y *jumped = false*.
- ***enemy_right_air***: necesita *isOnGround = false* e *isEnemy = right*.
- ***coin_mushroom_left***: necesita *isCoinMushroom = left*.
- ***coin_mushroom_up_jumped***: necesita *isCoinMushroom = up* y *jumped = true*.
- ***coin_mushroom_right***: *isCoinMushroom = right* y *jumped = false*.
- ***blocked_ground_jumped***: *isOnGround = true*, *isBlocked = true* y *jumped = true*
- ***blocked_ground***: *isOnGround = true*, *isBlocked = true* y *jumped = false*.
- ***blocked_air***: *isOnGround = false* e *isBlocked = true*
- ***enemy_left***: necesita *isEnemy = left*.
- ***normal***: necesita que no se dé ninguno de los casos anteriores.

La última cuestión que nos queda por afrontar sería:

- **Cuestión 3: ¿Cuál es el rango de celdas que ocupan los atributos *isEnemy*, *isCoinMushroom* e *isBlocked* para cada uno de sus valores?**

  Esto depende del atributo. Por un lado tenemos *isEnemy* e *isCoinMushroom*, que se comportan de manera análoga y por otro, *isBlocked*. Empezaramos por *isBlocked*, que es más sencillo.

  Caso *isBlocked*.

  Tras observar detenidamente el comportamiento de Mario en todos los bot que hemos creado a lo largo del curso, y viendo que decirle que está bloqueado cuando tiene un obstáculo a sus pies no es suficiente; hemos pensado que **este atributo tomará el valor *true* dependiendo de si Mario es grande o pequeño**.

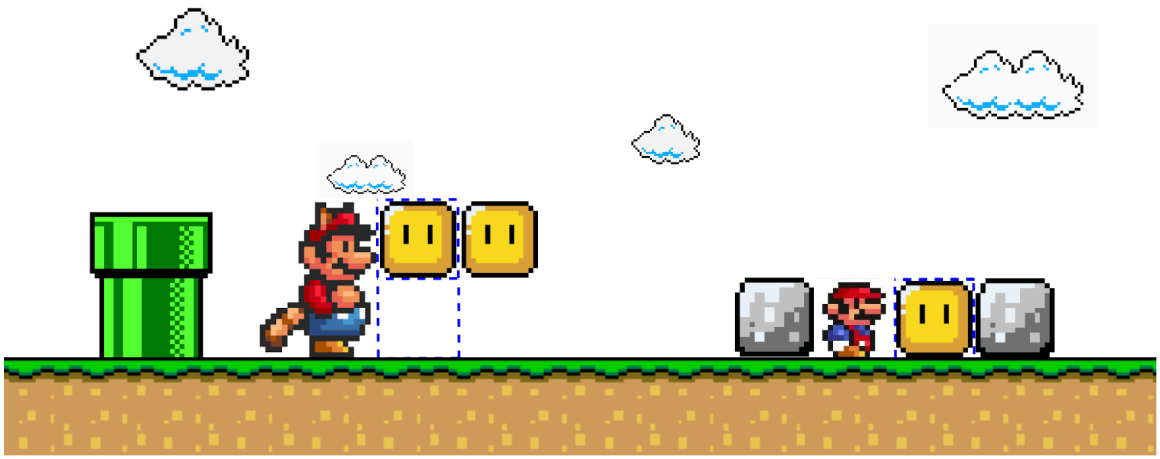

Figura 1: Región tenida en cuenta para determinar el valor de *isBlocked*.

Como se puede ver en la ilustración, es evidente que un Mario pequeño se quedará bloqueado en menos ocasiones, ya que se puede colar por huecos donde el grande no podría. Por ello, **necesitaremos comprobar si existe algún tipo de obstáculo sólo en las celdas a la derecha de Mario que se corresponden con su altura**, es decir, las celdas marcadas con azul en la ilustración.

Caso *isEnemy* e *isCoinMushroom*.

Este caso es bastante diferente al anterior, ya que estos atributos pueden tomar más valores que *isBlocked* y además **el área del mapa que estos abarcan depende de qué celdas puede alcanzar Mario en la ventana temporal que elijamos para evaluar su refuerzo**.

Como ya calculamos en la Práctica 2, Mario puede avanzar un máximo de 33 píxeles cuando la evaluación se ejecuta tras *n+6* ticks y 66 con *n+12*. Sin embargo, a diferencia de la práctica anterior, vamos a probar cómo resulta el comportamiento de Mario tras **probar una ventana temporal** mayor, **de *n+12* ticks**.

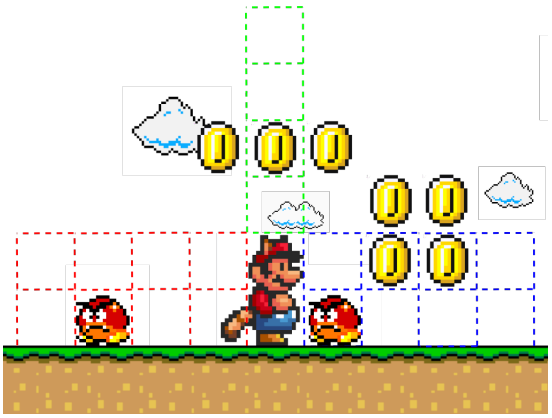

Figura 2: Región tenida en cuenta para determinar el valor de *isEnemy* e *isCoinMushroom* cuando Mario es grande.

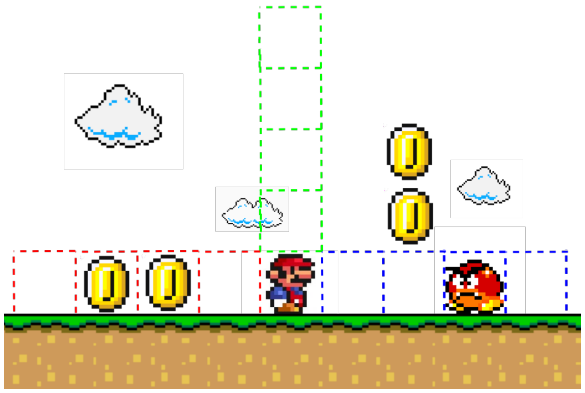

Figura 3: Región tenida en cuenta para determinar el valor de *isEnemy* e *isCoinMushroom* cuando Mario es pequeño.

Como se puede apreciar en la ilustración, **el área de influencia de estas variables es hasta 4 celdas** a su derecha para el valor *right* y otras 4 a la izquierda para el valor *left*. Esto se debe a que **4 celdas se corresponden con 64 píxeles**, es decir, dos píxeles menos de la distancia que puede alcanzar Mario en 12 ticks. La otra opción sería usar 5 celdas, pero esto puede hacer que le demos información a Mario de lugares que no va a poder alcanzar sólo por 2 píxeles. Asimismo, el número de celdas que encontramos en la ilustración para el valor *up* se ha medido de manera distinta, ya que es simplemente **el mayor número de celdas que Mario puede alcanzar tras un salto**.

Al igual que sucedía con *isBlocked*, **no es lo mismo que el agente sea pequeño que sea grande**, por lo que los valores *right* y *left* se adaptarán a su altura. Finalmente, podemos ver que no es necesario incluir las celdas diagonales respecto del agente, ya que estas se cubrirán por las que toman el valor *up* mientras se vaya moviendo por el mapa; por lo que nunca perderá esa información.

Finalmente, **necesitamos los atributos que nos servirán luego para evaluar el refuerzo que le otorgamos a cada una de las instancias**. Estos atributos se basan de nuevo en aquellos elementos clave que analizamos a la hora de definir nuestros estados. Por tanto, dichos atributos serían los siguientes:

- ***marioStatus***. Se trata de un atributo numérico que toma el valor 1 si Mario ha ganado una vida en el período de evaluación, -1 si la ha perdido y 0 si no se ha dado ninguno de los dos casos. Se trata del atributo que más impacto debe tener en la evaluación, ya que dijimos que la máxima prioridad de Mario es no perder vidas.
- ***distancePassedPhys***. Esta variable nos indica cuántos píxeles del mapa ha avanzado Mario horizontalmente durante el período de evaluación. Para evaluar la distancia recorrida por el agente también podríamos haber usado *distancePassedCells*, pero se trata de una variable mucho menos precisa (su unidad son las celdas, siendo una celda 16 píxeles). Este atributo sería el único que dispondríamos para evaluar el objetivo de maximizar la distancia recorrida.

  Como se puede observar, hemos decidido considerar las tres fuentes de *intermediate reward* posibles para nuestro modelo en lugar de tratarlo como una puntuación única. Esto hará que podamos darle la importancia que queramos a cada uno de ellos individualmente.
- ***killsByStomp***. Una manera de obtener *intermediate reward* es pisando a los enemigos. Por tanto, este atributo se encargará de medir cuántos enemigos ha derrotado Mario durante el período de evaluación. Sólo vamos a considerar las muertes por pisotón, debido a que las que se deben a usar bolas de fuego no nos aseguran que se puedan evaluar dentro de 12 ticks, ya que eliminan al enemigo en un tiempo mucho más variable que con los pisotones.
- ***coinsGained***. Este atributo se encarga de calcular cuántas monedas ha obtenido Mario durante el período de evaluación. Podríamos haber incluido aquí tanto monedas como champiñones, pero ambos no tienen la misma importancia; ya que el champiñón puede devolver vida al agente. Esto lo veremos en cuanto definamos la función de refuerzo.
- ***mushroomsDevoured***. Finalmente tenemos el atributo que obtiene el número de champiñones que ha conseguido Mario durante el período de evaluación.

Una vez descritos todos los atributos necesarios para calcular el refuerzo de las instancias, podemos elegir, al igual que hicimos con el atributo *situation*, si incluir el valor obtenido del refuerzo como un atributo más o calcularlo antes de introducir la instancia en la tabla Q. Siguiendo el mismo razonamiento que antes, **vamos a optar por guardarlo como atributo con el nombre de *reward***. Finalmente, su valor numérico se obtendrá a través de la fórmula del apartado siguiente.

4. Descripción de la función de refuerzo

La función de refuerzo es aquella que nos indica cómo de buena ha sido la acción elegida para el estado donde se encontraba el agente. Este resultado se obtiene a través de un valor numérico que tiene en cuenta los atributos que elegimos para la evaluación. Hemos decidido construir nuestra **función de refuerzo como un vector de pesos**. Estos pesos los vamos a distribuir en función de los objetivos que debe alcanzar Mario.

A pesar de que ambos objetivos son igual de importantes, **maximizar la *intermediate reward* es mucho más complicado que simplemente avanzar lo máximo posible en el mapa**. Es por ello que hemos decidido darle tanta importancia a recorrer la distancia máxima posible en 12 ticks como a conseguir una moneda o acabar con un enemigo. De esta forma **queremos intentar que Mario no ignore las monedas y enemigos que tiene a su alrededor**, como hacía el agente de la Práctica 2.

En el caso de que el número de píxeles recorridos dé un resultado negativo o directamente cero, como sabemos que no es bueno usar refuerzos negativos ni nulos; **inicializaremos todas las posiciones de la tabla Q a un valor ínfimo (0.1) para poder hacer uso de refuerzos nulos**. Si por ejemplo inicializáramos la tabla a 0, tendríamos que dar algún tipo de refuerzo en este tipo de situaciones, por muy pequeño que fuera. Sin embargo, muchos refuerzos positivos pequeños tienen mucho más impacto que un refuerzo pequeño fijo; que es lo que queremos evitar. De este modo, **el refuerzo mínimo que se puede obtener por la distancia avanzada durante el período de evaluación es de 0, aunque dicha distancia sea negativa**.

Por otro lado, **cada champiñón que obtenga Mario le dará un refuerzo equivalente al doble del que puede conseguir con una moneda o enemigo**, ya que aparte de que suma mucho más a la *intermediate reward*, puede darle vida cuando se encuentra pequeño.

Finalmente, **en el caso de que Mario pierda alguna vida durante el período de evaluación o su estado siga siendo el mismo, sumaremos 0 al refuerzo**. No debemos preocuparnos por los refuerzos nulos, ya que como dijimos antes todos los valores de la tabla Q van a estar inicializados a 0.1. Si se diera el caso de que Mario ha ganado vida tras recoger un champiñón, esto se recompensaría con un peso equivalente a recoger dicho champiñón; de modo que **aquellos champiñones que dan más vida a Mario valen el doble**.

La función de refuerzo quedaría así:

$$if\, marioStatus < 1$$

$$0{,}5 \cdot max(0, distancePassedPhys) + 33 \cdot killsByStomp + 33 \cdot coinsGained + 66 \cdot mushroomsDevoured$$

$$else$$

$$66 \cdot marioStatus + 0{,}5 \cdot max(0, distancePassedPhys) + 33 \cdot killsByStomp + 33 \cdot coinsGained$$
$$+66 \cdot mushroomsDevoured$$

5. Obtención de la tabla Q

Una vez planteado el agente que se va a encargar de la recogida de datos y que podemos encontrar en los ficheros **P3BotAgent.java** y las trece versiones de **P3HumanAgent.java** (una por cada uno de los estados que hemos definido para nuestro agente), dependiendo de si los controles del agente los lleva el bot que creamos para el Tutorial 1 o es controlado por el humano, respectivamente; podemos pensar sobre cómo implementar la tabla Q a partir de los datos de entrenamiento recogidos.

5.1. Implementación de la tabla Q

Todo el algoritmo de *QLearning* venía ya dado en el código entregado para el Tutorial 4. Sin embargo, este código necesitaba ser adaptado para este caso específico, por lo que hemos modificado parte de él.

Podemos encontrar el agente que aplica el aprendizaje por refuerzo (y con el que trabajaremos a partir de ahora) dentro del fichero **P3Agent.java**. A diferencia de los otros dos agentes que habíamos creado para la práctica, este no creará ningún tipo de archivo sobre el que guardar las instancias de entrenamiento; sino que simplemente **leerá de dichas instancias para construir la tabla Q en su constructor**.

Para ello, **se crea un nuevo objeto *QLearning_process* en el constructor del agente**. Este nuevo objeto consiste en una modificación sobre el *main* de la carpeta *qlearning* proporcionada en Aula Global para el Tutorial 4. Esta decisión en la implementación se toma teniendo en cuenta una recomendación del profesor, ya que esto abre las posibilidades de implementar on-line learning y decrementar el valor de α paulatinamente.

El proceso se inicializa teniendo en cuenta tanto las 7 acciones como las 13 situaciones descritas anteriormente. El constructor del objeto *QLearning_process* estará a cargo de **leer el fichero de datos** (evitando las 18 primeras líneas destinadas al *header* de Weka) y **añadir al mapa de estados las instancias del fichero como tuplas** en forma de:

$$(s, a, s', R)$$

$$s = Estado\ actual,\ a = Acción\ ejecutada,\ s' = Estado\ alcanzado,\ R = Refuerzo\ obtenido$$

Para codificar las situaciones y acciones en forma de número como requiere el algoritmo *QLearning* proporcionado, se hace uso de los métodos ***situation_toNumber*** y ***action_toNumber***. Ambos consisten en un simple switch para **asignar a cada situación o acción un valor numérico**.

El siguiente paso a seguir consiste en aplicar *Qlearning* propiamente dicho para obtener la tabla Q. El método ***qlearn()*** es el que está a cargo de ello. Este método venía dado en el código de ejemplo, pero **lo modificamos para nuestro problema específico**:

- La primera diferencia con el método original es que el nuestro **devuelve la tabla Q que calcula** para que la podamos usarla siempre que queramos en P3Agent.
- También **eliminamos la definición de estado inicial y final**, ya que en nuestro problema estos no existen como tal: Mario no tiene por qué comenzar o terminar en un estado específico de los que hemos definido.
- Por último, **quitamos la parte del código que se encargaba de imprimir por pantalla las transiciones que había de un estado a otro para llegar desde el estado inicial al final**, ya que dicho recorrido no existe para nuestro problema como indicamos en el punto anterior.

Dentro de este método también **se definen los parámetros que usamos para construir nuestra tabla Q**. Todos ellos han sido adaptados a esta práctica:

- **Los valores de la tabla Q son inicializados a 0.1** en lugar de a 0, tal y como indicamos anteriormente, con el objetivo de poder usar refuerzos nulos.
- **El valor de α va decreciendo paulatinamente**, es decir, deja de ser un valor estático para ser un valor dinámico. Su valor va disminuyendo según va leyendo nuevas instancias dentro del fichero de entrenamiento, de modo que aquellas que se encuentran al final tienen un impacto mínimo en la construcción de la tabla Q. Decidimos que sería bueno hacerlo de esta manera porque es cómo se recomienda tratar el valor de α cuando se está tratando con **un problema no determinista**. Por tanto, debido a que el valor de α asume el determinismo del sistema cuando este es cercano a 1, su valor base no puede ser muy elevado. Decidimos entonces que 0,5 sería una buena opción. Para obtener el valor específico de α de una tupla en concreto, sólo tenemos que remitirnos al método ***getAlpha()***, cuya fórmula es la siguiente:

$$0,5 - \frac{instance}{tot\_instance}$$

$$instance = Instancia\ actual,\ tot\_instance = Instancias\ totales$$

Finalmente, para poder aplicar este valor variable de α a cada una de las tuplas que se van añadiendo para construir la tabla Q, **modificamos también el método *actualizarTablaQ()***. Para ello, incluimos α como nuevo parámetro y lo utilizamos dentro del método para actualizar el valor dado a α en el constructor de *Qlearning*.

- En cuanto al **valor de** γ, en nuestro caso **debe ser muy pequeño; ya que afecta a la propagación de los Q valores**. Este hecho hace que se otorgue un refuerzo positivo a aquellos conjuntos de situaciones y acciones que, aun no siendo los más adecuados, van sucedidos por acciones mucho más idóneas que ocurren dentro de la ventana temporal de evaluación. Esta es la razón por la que γ tiene un valor tan bajo: 0.05.
- Finalmente, **el número máximo de ciclos es 1** porque probando con otros números obteníamos una tabla Q con valores cercanos al 0 bastante extraña.

Tras obtener la tabla Q, la imprimimos en pantalla antes de terminar con el constructor de P3Agent haciendo uso del método ***mostrarTabla()***.

5.2. Obtención del conjunto de entrenamiento

Como ya indicamos con anterioridad, nuestro **método de entrenamiento** para el agente de esta práctica está basado en un **sistema mixto bot y humano**. Tomamos esta decisión tras ver que las instancias el bot del Tutorial 1 enseñaba mucho mejor a Mario cómo debía jugar que jugando nosotros. Con esto en mente, creamos un nuevo bot, **P3BotAgent**, que se mueve por el mapa ejecutando la acción que hemos considerado óptima para cada uno de los estados que hemos incluido. Estas acciones han sido definidas a partir de varias rondas de experimentación probando distinto valores; ya que es difícil saber cuál es la mejor de antemano. Así, su **abanico de acciones** quedó determinado de la siguiente manera:

- Para la situación ***blocked_ground_jumped***, la acción más adecuada es quedarse quieto, para que así pueda saltar el obstáculo en el tick siguiente.
- Para la situación ***blocked_ground***, la acción más adecuada es avanzar saltando, ya que sólo saltar hace que Mario caiga en la misma posición donde estaba bloqueado.
- Para la situación ***blocked_air***, la acción más adecuada es avanzar saltando también, para que se impulse aún más en el aire y pueda saltar obstáculos más altos.
- Para la situación ***enemy_up***, la acción más adecuada es moverse hacia la derecha, para evitar al enemigo y seguir avanzando.
- Para la situación ***enemy_right_ground_jumped***, la acción más adecuada es moverse hacia la derecha, para poder saltar en el tick siguiente al enemigo. No hemos visto que acercarse al enemigo durante un solo tick sea un peligro y además hemos comprobado que esto ayuda a que el agente acabe con ellos mediante un pisotón.
- Para la situación ***enemy_right_ground***, la acción más adecuada es avanzar saltando, ya que no hay otra manera de evitar a un enemigo (un salto a secas sólo consigue que se encuentre al enemigo de frente cuando llega al suelo).
- Para la situación ***enemy_right_air***, la acción más adecuada es avanzar saltando de nuevo, para evitar encontrarnos con el enemigo.
- Para la situación ***enemy_left***, la acción más adecuada es moverse a la derecha, ya que dijimos que no íbamos a intentar acabar con los enemigos que no obstaculizaran a Mario.
- Para la situación ***coin_mushroom_left***, la acción más adecuada es moverse a la izquierda, para recoger la moneda o champiñón.

- Para la situación ***coin_mushroom_up_jumped***, la acción más adecuada es quedarse quieto, para saltar a recogerlas en el siguiente tick.
- Para la situación ***coin_mushroom_up***, la acción más adecuada es avanzar saltando; ya que si salta sin avanzar y la moneda se encuentra tras algún bloque, Mario se va a quedar saltando indefinidamente para intentar conseguirla.
- Para la situación ***coin_mushroom_right***, la acción más adecuada es moverse a la derecha, para poder recoger las monedas o champiñón.
- Para la situación ***normal***, la acción más adecuada es moverse a la derecha. Así nuestro agente seguirá avanzando en el nivel.

Una vez que comprobamos que estas eran las acciones más adecuadas para cada uno de los estados, vimos que **nuestro agente se quedaba bloqueado en algunas situaciones debido a cómo hemos establecido las prioridades entre los distintos estados**. Este problema ocurre principalmente cuando Mario se encuentra bloqueado por algún tipo de obstáculo pero tiene a su vez monedas a su derecha. Como el segundo estado tiene prioridad sobre el primero, el agente intenta avanzar sin éxito. Debido a esta problemática, decidimos incluir un **contador de bloqueo, *counter_blocked***. Cuando este contador llega a un número lo suficientemente alto (en nuestro caso 5 ticks), la acción que se ejecuta por defecto es ***jump_right_long***; para saltar el obstáculo que mantenía a Mario en el mismo sitio. Esta **acción** es **durativa**, ya que **ejecuta *jump_right* durante dos ticks**, los suficientes para desatascar a Mario.

Para obtener una tabla Q adecuada, sólo tuvimos que poner al bot a jugar un nivel que se pasara. Dicho nivel se trata de aquel que tiene semilla 8 y dificultad mínima. **Las instancias que recogimos durante esta primera fase de entrenamiento se encuentran en *entrenamiento_bot.arff***.

La segunda fase del entrenamiento se ha centrado más en **recoger instancias para cada una de los estados y acciones que no cubría el bot**, es decir, aquellas que no eran las más adecuadas para cada caso.

Para obtener las instancias con el humano de una manera más controlada **se han creado 13 subagentes humanos**. Cada uno de ellos está a cargo de grabar las instancias de una situación para las acciones que no cubre el bot. Los agentes se denominan *P3HumanAgentx* donde x es el número de la situación a la que se refieren. Con esta medida logramos controlar y minimizar los daños que resultan de grabar todas las instancias del humano de golpe, ya que somos muy propensos a errores y estos puede afectar negativamente a la tabla Q. Se jugó sólo en las semillas 0 y 1 del nivel de dificultad más sencillo para obtener estas instancias.

Del proceso de jugar con estos agentes se obtienen 13 archivos. Los datos recogidos en estos archivos se agregan a los recogidos con el bot en un fichero denominado *base_conocimiento_mixed*. Teniendo en cuenta que aplicamos un valor de α **descendiente**, la unión de los datos del bot y el humano es **mezclada de manera aleatoria**. La decisión de llevar esta acción está basada en que para un valor descendiente de α, las instancias que se leen más tarde, cuando se crea la tabla Q, pierden relevancia. De esta manera **conseguimos una mezcla de instancias homogénea y libre de relaciones de orden**.

El proceso de mezcla aleatoria se lleva a cabo utilizando la herramienta Weka. Los pasos a seguir son:

$Explorer \rightarrow Openfile \rightarrow filter \rightarrow choose \rightarrow filters \rightarrow unsupervised \rightarrow instance \rightarrow Randomize \rightarrow Seed : 42.$

5.3. Interpretación de la tabla Q obtenida

La tabla Q que hemos obtenido tras las la primera fase de recogida de instancias, es decir, la del bot, es la siguiente:

	a0	**a1**	**a2**	**a3**	**a4**	**a5**	**a6**
s0	**55.99**	0.1	0.1	0.1	0.1	0.1	0.1
s1	0.1	0.1	0.1	**55.04**	0.1	0.1	0.1
s2	0.1	0.1	0.1	**79.72**	0.1	0.1	0.1
s3	**147.64**	0.1	0.1	0.1	0.1	0.1	0.1
s4	0.1	0.1	0.1	0.1	0.1	0.1	0.1
s5	0.1	**40.5**	0.1	0.1	0.1	0.1	0.1
s6	0.1	0.1	0.1	**71.9**	**38.2**	0.1	**111.04**
s7	0.1	0.1	0.1	**135.41**	0.1	0.1	0.1
s8	**87.9**	0.1	0.1	**64.54**	**70.26**	0.1	0.1
s9	0.1	0.1	0.1	**5.1**	0.1	0.1	**13.45**
s10	0.1	0.1	0.1	**161.41**	0.1	0.1	0.1
s11	0.1	0.1	0.1	**159.73**	**8.72**	0.1	0.1
s12	**27.62**	0.1	0.1	0.1	**27.31**	0.1	0.1

Cuadro 1: Tabla Q obtenida tras la primera fase de entrenamiento.

Tras analizarla detenidamente, podemos hacer las siguientes observaciones al respecto:

- La situación s4, que se corresponde con ***enemy_up*, no se ha dado durante todo el entrenamiento**. Sin embargo, su acción más adecuada es moverse a la derecha (a0), que es la misma acción que se elige cuando no se ha rellenado ningún valor para una situación (por ser la primera); y por eso no lo notamos al probar el agente que controla P3Agent.
- Las situaciones s1, s2, s3, s5, s7 y s10; que se corresponden con ***enemy_right_ground_jumped*, *enemy_right_ground*, *enemy_right_air*, *enemy_left*, *coin_mushroom_left*, *coin_ mushroom_up*** y ***blocked_ground*** respectivamente, sólo contienen Q valores para la acción que le indicamos al bot que era la válida en esas situaciones. Suponemos que se trata de situaciones no conflictivas a la hora de quedarse bloqueado.
- Sin embargo, las situaciones s6, s8, s9, s11 y s12; que se corresponden con ***coin_mushroom_up_ jumped*, *coin_ mushroom_right*, *blocked_ground_jumped*, *blocked_ground_air*** y ***normal*** respectivamente, tienen Q valores para al menos dos acciones. En algunas de ellas esto se debe a que se rellena la acción que le indicamos hacer al bot y la acción durativa (a4); y en otros casos simplemente se graban más acciones de las que se le indicó hacer al bot (s6 y s8). Sin embargo, **la acción que indicamos al bot que debía hacer es la que tiene un mayor Q valor para cada estado**; por lo que el asunto anterior pasa a ser secundario. Además, **salvo en s6, la segunda acción con un Q valor mayor es la acción durativa (a4)**; por lo que los resultados son satisfactorios (s6 o *coin_mushroom_up_ jumped* es un estado que tiene un tick de duración, por lo que no necesita esa acción). Decimos esto porque nuestro objetivo a la hora de elegir acciones es escoger la segunda mejor cuando el contador de bloqueo llegue a 5, al igual que hacíamos con el bot.

La tabla Q final, obtenida tras las dos fases de entrenamiento (tanto con el bot como con el humano) resulta ser la siguiente:

	a0	a1	a2	a3	a4	a5	a6
s0	**37.07**	**2.11**	**1.86**	**21.07**	**0.1**	**2.39**	**0.1**
s1	**12.33**	**2.4**	**1.34**	**48.17**	**0.1**	**2.45**	**2.92**
s2	**25.91**	**3.56**	**7.4**	**62.23**	**0.1**	**2.27**	**4.36**
s3	**116.75**	**4.01**	**6.06**	**15.74**	**0.1**	**2.32**	**6.69**
s4	**1.64**	**2.48**	**4.51**	**9.37**	**0.1**	**1.9**	**1.99**
s5	**16.98**	**121.53**	**7.22**	**50.16**	**0.1**	**6.97**	**6.84**
s6	**48.73**	**0.1**	**7.24**	**71.51**	**39.61**	**4.96**	**83.27**
s7	**22.42**	**11.44**	**41.52**	**76.77**	**0.1**	**38.49**	**55.63**
s8	**196.27**	**7.67**	**4.83**	**72.53**	**71.73**	**6.55**	**7.27**
s9	**1.32**	**2.65**	**1.74**	**5.23**	**0.1**	**2.41**	**30.09**
s10	**1.93**	**2.02**	**3.9**	**37.93**	**0.1**	**1.71**	**2.02**
s11	**3.14**	**1.72**	**3.17**	**63.86**	**9.79**	**2.29**	**6.34**
s12	**33.13**	**1.94**	**2.15**	**20.68**	**37.61**	**2.92**	**1.96**

Cuadro 2: Tabla Q final.

Como podemos apreciar en la tabla, las acciones y situaciones que el bot no llegaba a cubrir son completadas por el humano. Dicho agente es perfecto para esta tarea, ya que se tratan de acciones sub-óptimas para la situación correspondiente y de esta manera podemos utilizar a nuestro favor el error, debido su excesivo tiempo de reacción.

Marcados sobre un fondo amarillo podemos ver el valor más alto para cada una de las situaciones. Predominan claramente las acciones aprendidas por el bot ya que son las más precisas. En el caso de la situación 4, debido a que esto es una situación que nunca se llegó a dar con el bot, aprende del humano que, si tiene un enemigo encima el mejor resultado es saltar a la derecha. Parece ilógico en un primer momento, pero tras experimentar hemos visto que si tiene un enemigo encima no hay forma de evitar la colisión, ni siquiera agachándose o huyendo. Así, el agente se resigna a recibir el daño del enemigo, pero escoge la acción de saltar a la derecha porque de esta manera puede recibir cierta recompensa al recorrer distancia y, debido a que estos enemigos suelen caer desde un obstáculo alto, consigue sobrepasar dicho obstáculo. En pocas palabras, dentro de lo malo, escoge lo que más le beneficia.

Otro caso es el de la situación 12, *normal*, cuya acción con mejor Q valor cambia de moverse a la derecha a la acción durativa de avanzar saltando. Tiene bastante sentido que esto sea así, ya que con ambas opciones se consigue avanzar prácticamente lo mismo, pero con la segunda se evita mucho mejor a los enemigos. Otro punto a destacar es que la diferencia entre ambas ya en la tabla Q del bot no era muy significativa.

En la tabla podemos ver algunos Q valores correspondientes a la acción 4 todavía con su valor de preinicialización (0.1). Esto se debe a que para los agentes humanos no fue posible identificar la acción *right_jump_long* y los saltos largos hacia la derecha se consideran igualmente saltos a la derecha.

6. Obtención del agente automático

El objetivo del agente automático de esta práctica no es más que identificar en qué situación se encuentra para saber así qué acción tiene que ejecutar fijándose en exclusiva de la tabla Q de la que dispone.

6.1. Pertenencia a un estado

La pertenencia a un estado se concluye de igual forma que en el bot o en el humano, utilizando el método ***getState()***. Este método recibe en el caso del humano y el bot los argumentos desde la línea que va a ser escrita en el fichero mientras que en el agente final los recibe a través de las variables globales utilizadas para la observación.

Los argumentos del método *getState()* son *isOnGround*, *isEnemy*, *isCoinMushroom*, *isBlocked*, y *jumped*. Dependiendo del valor de estos argumentos el método concluye la situación en la que se encuentra. Además, se ciñe a la prioridad entre detección situaciones anteriormente explicadas para asignar una situación u otra en el caso de que pertenezca a varias.

6.2. Toma de decisiones

Finalmente, una vez se ha calculado la tabla Q, el agente, en el momento de determinar la acción a ejecutar, consulta la tabla para obtener la mejor acción dado el estado actual en el que se encuentre. **Para determinar la mejor acción hace uso del método *obtenerAccion()***, que forma parte del algoritmo *qlearning* proporcionado para esta práctica.

Sobre esta funcionalidad básica se implementa un argumento extra para dicho método. Este **argumento, *mode***, se utilizará para determinar si el método **devuelve la mejor acción (*mode=1*) o la segunda mejor acción (*mode=0*)**. Esta mejora se implementa para que, en caso de detectarse que Mario no avanza en el mapa durante 5 ticks, se ejecute la segunda mejor acción para desbloquearlo.

6.3. Funcionamiento del agente automático

Lo que más nos llamó la atención al ver por primera vez el comportamiento del agente final de esta práctica, fue que sus saltos eran muy largos y amplios; como el agente que presentamos en la Práctica 1. Esto se debe a que, como ya explicamos antes, aplica la acción durativa de avanzar saltando durante dos ticks siempre que se dé el estado *normal*.

Sin tener en cuenta esta anécdota, el comportamiento general de Mario se basa en avanzar hacia delante (aunque a veces da saltos prolongados cuando el mecanismo del salto le es viable y se encuentra en el estado *normal*. Cuando se encuentra con enemigos a la derecha, los salta con bastante antelación, consiguiéndolos evitar la mayoría de las ocasiones. Por contra, los enemigos a la izquierda los evita siempre.

En cuanto a las monedas, siempre intenta recogerlas mientras que se encuentre en su rango; por lo que es bastante común verlo irse hacia detrás a recoger monedas o champiñones y saltar a por ellas cuando es pequeño. Como dato curioso, a veces trata de saltar a por monedas que se encuentran encima de bloques, por lo que sólo consigue romperlos y sigue avanzando.

Uno de los mayores problemas que residen en este agente es que debido a los grandes saltos que ejecuta, la mayoría de las veces que resulta herido cuando va a llegar al suelo. Por otro lado, hay algunos obstáculos que es incapaz de superar, sobre todo si estos son de gran altura. Sin embargo, hemos conseguido que pase a través de dos bloques, algo que ninguno de los agentes que habíamos creado había conseguido.

7. Evaluación de los resultados

Hemos comparado nuestro agente con el *BaselineAgent, T1Agent, P1Agent* y *P2Agent*, comparando tanto la distancia recorrida como el refuerzo conseguido en semillas nunca antes entrenadas (de la 1001 a la 1026).

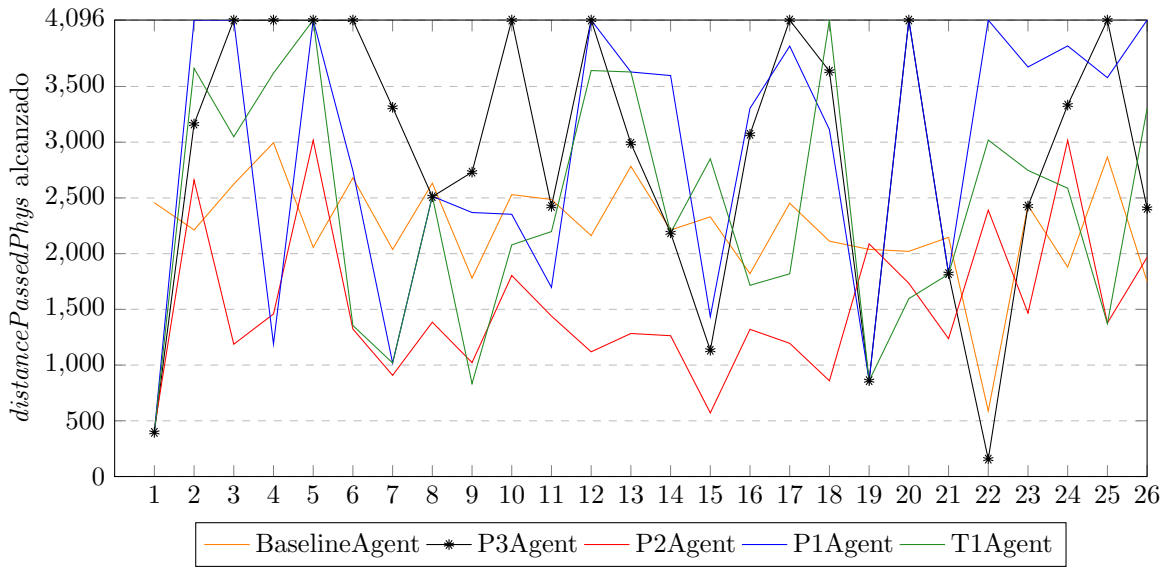

Figura 4: Comparación de agentes en semillas nunca antes entrenadas (**distancia recorrida**)

Se puede observar que el agente es bastante bueno, consiguiendo mejorar el récord de victorias que había conseguido el bot de la Práctica 1. Sin embargo, consigue peores resultados en algunos niveles por los problemas expresados en el apartado anterior (se choca bastante con los enemigos por sus grandes saltos y además se queda bloqueado en algunos obstáculos que los otros bot superan sin problemas).

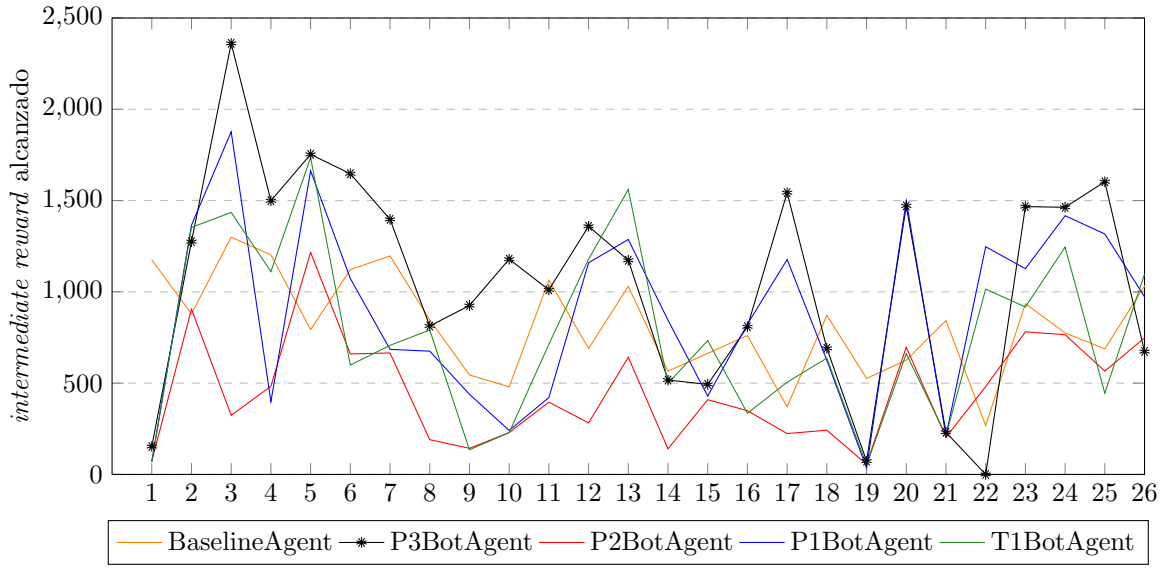

Figura 5: Comparación de agentes en semillas nunca antes entrenadas (**recompensa**)

En cuanto al refuerzo que obtiene, es bastante mejor que los demás. Esto se debe a que es capaz de pasarse muchos niveles, en los que además se centra en recoger monedas, algo que no hacían ninguno de los otros bot que estamos evaluando. Aquellos niveles en los que obtiene un refuerzo muy bajo son en los que recorrió poca distancia, por lo que apenas pudo aumentar su refuerzo.

8. Conclusiones

Las principales conclusiones que extraemos tras completar la práctica son las siguientes:

- La decisión de los estados a definir debe tener en cuenta las posibles situaciones en las que Mario se puede encontrar así como el orden de prioridad a la hora de detectarlas. No es lo mismo tener una moneda delante que tener tanto una moneda como un enemigo. Ya que queremos sobrevivir el máximo tiempo posible.
- La definición de acciones a lo largo de un breve periodo de *ticks* puede resultar muy útil a la hora de mejorar el comportamiento del agente y así fue en nuestro caso con la acción de salto largo a la derecha *right_jump_long* aplicada a la situación 9 (*normal*:ni enemigos, ni monedas ni obstáculos).
- A la hora de aplicar *Qlearning* es muy importante comprender el significado de los parámetros α y γ.

  En primer lugar, α está relacionado con el determinismo que se supone en el sistema, siendo 1 si el sistema es completamente determinista. Por esta razón en nuestro caso es 'relativamente' bajo con un valor de 0.5.

  En segundo lugar, γ está relacionado con la propagación hacia atrás. Debido a que usábamos una ventana temporal de medio segundo y a que la propagación en este dominio no era siempre intuitiva, decidimos asignar siempre a este parámetros valores bajos (0.05).
- Sobre α también sabíamos que es una práctica muy común reducir este valor conforme avanza la creación de la tabla Q. Sin embargo, esto tiene una consecuencia que aprendimos más

adelante, durante la realización de la práctica. Reducir el valor de dicho parámetro conlleva una depreciación de las instancias que son leídas al final de la creación de la tabla Q. Esto quiere decir: cuanto antes se lea una instancia, más importante será su contribución. Para solventar el problema que puede ocasionar grabar las instancias de manera secuencial hicimos pasar dichas instancias por el *Randomizador* de Weka hasta obtener una mezcla aleatoria con la que estábamos satisfechos. Esto proporcionaba a la muestra homogeneidad y la liberaba de las relaciones de orden establecidas al jugar un nivel y guardar las instancias secuencialmente.

- Se ha de evitar el uso de refuerzos negativos o iguales a 0 ya que no suelen dar buenos resultados. Sin embargo, preinicializando la tabla a un valor muy bajo podemos asignar refuerzos de valor 0 y neutralizar el impacto de una acción sub-óptima sobre la tabla producida. Medida que llevamos a cabo.

9. Comentarios personales

Esta práctica ha sido muy útil, no solo a la hora de aclarar conceptos sino también para poder ver uno de los algoritmos estudiados en clase en acción.

En primer lugar, creemos que gracias al tutorial anterior obtuvimos un gran conocimiento acerca de como funciona *Qlearning* y, junto con esta práctica creemos haber asimilado los conceptos claramente. En esta asignatura las lecciones no siempre se pueden aprender a primera vista ya que existen muchos factores a tener en cuenta y se ha de conocer cada uno de ellos. Prácticas como esta nos ayudan a familiarizarnos con estos factores así como aprender de los problemas relacionados con el uso de un algoritmo determinado así como pequeños trucos y matices que seguramente nos serán de gran ayuda en el futuro (más inmediatamente, el examen final).

En segundo lugar, la práctica nos muestra el poder del algoritmo implementado así como su evolución. Poco a poco fuimos refinando nuestro conjunto definido de situaciones y de acciones además de añadir extras como el decremento de α y la mezcla aleatoria de las instancias recogidas. Todos estos cambios tuvieron un impacto directo e inmediato sobre el funcionamiento y pudimos comprender fácilmente qué entorpecía o mejoraba dicho funcionamiento.

En definitiva, un práctica muy completa que creemos indispensable para esta asignatura.