Aprendizaje Automático
Práctica 2

García Fernández, Antonio 100346053
García García, Alba María 100346091

Leganés G83

17 Abril 2018

Índice

1. Introducción **2**

2. Balanceo de instancias **2**

3. Descripción de los atributos seleccionados **3**

4. Creación de la base de conocimiento **10**

5. Tratamiento sobre los datos **16**

6. Estructuras de datos utilizadas **17**
6.1. Fase de recogida de instancias de la base de conocimiento 17
6.2. Fase de *Instance Based Learning* 18

7. Funciones del *Instance Based Learning* **19**
7.1. Función de pertenencia 19
7.2. Función de similitud 20
7.3. Función de evaluación 22

8. Evaluación de los resultados **24**

9. Conclusiones **26**

10. Comentarios personales **27**

1. Introducción

En esta segunda práctica se planteaba la creación de un agente *P2BotAgent* que, haciendo uso de técnicas de **IBL (*Instance Based Learning*)**, fuera capaz de avanzar por los diferentes niveles así como de intentar incrementar su puntuación.

Este documento presenta las acciones y decisiones tenidas en cuenta tanto, a nivel de recogida de datos, como de implementación, a lo largo de la realización de la tarea propuesta.

2. Balanceo de instancias

Un **problema muy marcado** de este simulador de *Super Mario Bros*, *MarioAI*, es el **gran desbalanceo de situaciones** que presenta: en la mayoría de las veces, Mario se limita a correr y no son comunes aquellas situaciones que incluyen recoger una moneda o cualquier otro tipo de objeto que aumente la *Intermediate reward*, como los campiñones; al igual que aquellas que incluyen matar o ser herido por un enemigo.

Teniendo en cuenta que el **objetivo** de Mario al pasarse los niveles es **avanzar lo máximo posible** y **maximizar la *Intermediate reward***; no nos sirven las instancias que podamos obtener de un agente cualquiera sin hacer algún tipo de filtro o selección.

A la hora de discernir estas situaciones, **no creemos que sea necesario tratar de buscar situaciones muy específicas**, ya que eso puede propiciar que a cada una le corresponda una única acción, lo que tiraría por tierra la función de evaluación, ya que esta sirve para elegir la mejor acción de entre las instancias más similares que se encontraron de la misma situación. La **función de similitud** ya se encargará de **diferenciar distintos casos** dentro de una misma situación.

Para ello, hemos considerado **cuatro situaciones clave** para la consecución de estos dos objetivos:

- **Mario tiene una moneda o champiñón que da vida *cerca*.**
  Ambos objetos tienen el mismo tratamiento porque siempre dan puntos al *Intermediate reward* cuando son recogidos. En este grupo también puede entrar la flor de fuego, pero estas aparecen tras golpear ciertos bloques, al igual que los champiñones, pero a diferencia de ellos no se mueven en horizontal, por lo que requeriría que Mario se subiera a los bloques y las recogiera. Hemos considerado este comportamiento complicado en exceso como para que lo pueda aprender usando *Instance Based Learning*, y esta es la razón para no incluir las flores de fuego. Tampoco consideramos los bloques ? porque no siempre dan *reward*, por lo que puede confundir más a Mario que ayudarlo a aprender. La palabra *cerca* adquiere aquí el significado de 'están lo suficientemente cerca como para que Mario pueda obtener esa moneda o champiñón en el número de ticks futuros elegidos para la función de evualuación (*n+12* o *n+24*)', ya que sólo de este modo obtener dicho objeto sería evaluable; y por tanto, útil. Calcularemos el número de celdas que Mario puede recorrer en *n+12* o *n+24* ticks a la hora de implementar esta situación.
- **Mario tiene un enemigo *cerca*.**
  Esta situación es completamente distinta a la anterior, ya que Mario puede tanto aumentar como disminuir la *Intermediate reward* cuando se encuentra con un enemigo. Sin embargo, sí que es analóga a la situación anterior en el sentido de que la palabra *cerca* adquiere el mismo significado.

- **Mario tiene un obstáculo delante**.
  Esta situación es necesaria para que Mario pueda avanzar, ya que los obstáculos son lo único que puede bloquearle. En este caso no hacemos uso de la palabra *cerca*: a pesar de que en las dos situaciones anteriores tenía sentido mirar la posición del objeto clave para la situación (moneda, champiñón o enemigo), para avanzar basta con moverse a la derecha y saltar. Por tanto, sólo necesitamos mirar la casilla del mapa justo en frente de Mario en este caso.

- **Mario no tiene un obstáculo delante**.
  Incluimos este caso para poder cubrir todas las situaciones en las que Mario se puede ver envuelto, ya que todas las instancias que recibamos en tiempo real tienen que clasificarse en algún grupo. Esta situación es muy heterogénea y las diferentes acciones que se pueden tomar depende en gran medida de los elementos que hayan o no alrededor de Mario que puedan influir en el *Intermediate reward*.

El objetivo es programar estas situaciones a través de contadores, de modo que **grabemos en los ficheros de entrenamiento el mismo número de instancias de cada una de las situaciones**. Así, la recogida de datos con los agentes será realmente útil para construir la base de conocimiento que necesita Mario para aplicar el *Instance Based Learning*.

En el caso de que **una instancia concreta pudiera pertenecer a más de una de las situaciones anteriores**, (no todas son todas mutualmente excluyentes) la encasillaremos en una de ellas, utilizando **relaciones de precedencia**. Hemos priorizado las situaciones en función de su probabilidad para aumentar la *Intermediate reward*, ya que es mucho más complicado que avanzar en el mapa. De este modo, el orden sería el mismo en el que se han descrito las situaciones previamente.

3. Descripción de los atributos seleccionados

Los atributos que hemos pensado, y que por tanto se incluirán en la **base de conocimiento**, son aquellos que se utilizarán para el balanceo de instancias y las funciones de similitud y evaluación. También incluiremos la acción que ejecuta Mario en el tick actual, el resultado de la función de evaluación y la situación a la que pertenece la instancia.

A la hora de crear estos atributos hemos tenido en cuenta que la **visión del dominio** que posee Mario es **muy limitada**, ya que debemos evaluar las acciones en un rango máximo de un segundo. Esto produce que debamos pensar en atributos con los que podamos evaluar una ***Intermediate reward* inmediata**, no distendida en el tiempo.

Debido a que debemos ejecutar el *Instance Based Learning* en **tiempo real**, hemos decidido **calcular el resultado de la función de pertenencia y de evaluación durante la fase de recogida de instancias de entrenamiento** e incorporar dichos resultados como atributos. Con respecto a la función de similitud, no es adecuado incorporarla durante esta fase: eso corresponde a la fase de *IBL*, en la que el bot cuenta con una base de conocimiento con la que tomar decisiones.

- ***situation***. La **función de pertenencia** debe identificar la situación a la que pertenece la instancia que se está evaluando. Como estas situaciones son las mismas que definimos en la sección anterior, podemos obtener el valor de este atributo mientras vamos comprobando en el balanceo de instancias a qué situación pertenece cada una. Así, los valores que puede tomar son los correspondientes a las situaciones identificadas: *coin_mushroom*, *enemy*, *blocked* y *no_blocked*.

- ***evaluation***. En cuanto a la **función de evaluación**, podemos obtener los parámetros necesarios para calcular su valor durante la recogida de instancias. Por tanto, no es necesario posponer su cálculo a la ejecución de la toma de decisiones en tiempo real, ya que estaríamos repitiendo los mismos cálculos cada vez que la misma instancia es seleccionada por la función de similitud. Calculando su valor antes logramos reducir ese cálculo a una única vez para cada tick y guardarlo como atributo. Además, estas operaciones consumen tiempo valioso de la ejecución del *Instance Based Learning*, siendo el tiempo para elegir la acción de Mario en cada tick crítico. La forma de evaluar será a través de un valor numérico, que indicará una mayor idoneidad cuanto más alta sea, existiendo la posibilidad de evaluaciones negativas.

Los atributos necesarios para el balanceo de instancias son los mismos que usamos en para la **función de similitud** y para las de **pertenencia**. Estos atributos son los siguientes:

- ***isEnemy***. Este atributo puede tomar cuatro valores distintos, que nos indican la posición del enemigo con respecto a Mario (en el caso de que lo hubiera) o si directamente este no existe (valor *none*). Estas tres regiones en las que se puede encontrar el enemigo se pensaron teniendo en cuenta las posiciones en las que Mario se puede mover: delante, detrás y arriba; a las que les corresponden los valores *front*, *behind* y *up*. [1] No incluimos el valor *down*, ya que esta situación sólo se da cuando Mario está en el aire y tiene un enemigo debajo (al que aplastará o no), por lo que habrá saltado para matarlo cuando lo tiene delante o detrás o simplemente es un hecho fortuito. Los dos primeros casos los cubrimos con los valores *front* y *behind*. Sin embargo, el último no consideramos que sea buena idea incluirlo, ya que no existe una razón para conseguir esta *Intermediate reward*, sino que es causa de la suerte, por lo que la acción que lo produjo no debería ser evaluada positivamente por ello. Una vez explicados los posibles valores que puede tomar esta variable, podemos ver por qué es útil tanto para la función de pertenencia como para la de similitud:

  - En el caso de la **función de pertenencia**, sirve para averiguar si Mario se encuentra en la situación de *Mario tiene un enemigo cerca*, comprobando que hay enemigos a los alrededores (un valor diferente a *none*).
  - Para la **función de similitud**, son los otros tres atributos los que necesitamos: *front*, *behind* y *up*. Así podremos discernir dónde se encuentra el enemigo y por tanto, qué acción debería tomar Mario para evitarlo o intentar eliminarlo.

- ***isCoinMushroom***. Al igual que *isEnemy*, este atributo puede tomar cuatro valores: *front*, *behind*, *up* y *none*[2]; con la diferencia de que en lugar de valorar un elemento clave como es el enemigo, valora por igual monedas y champiñones. La razón para no tomar el valor *down* es el mismo motivo que para el atributo anterior. En un principio se iba a crear una variable para cada uno de los elementos, pero ambas toman el mismo código dentro de la matriz de *Merged Observation*, para el nivel de detalle 1 (código 2). Es cierto que si aumentamos el detalle podríamos discernir entre ambas, pero esto dificultaría mucho la detección del enemigo para *isEnemy*, ya que con ese nivel de detalle sólo hay que comprobar el código 80, mientras que si el nivel de detalle es 0 hay que comprobar un código por cada tipo de enemigo. De todas formas, esto no es ningún problema, ya que todos los enemigos a los que nos enfrentamos son *goombas*, por lo que no hay necesidad de diferenciarlos. Además, las monedas y los champiñones se tratan de manera diferenciada en la función de evaluación.

  - Este atributo es útil por el mismo motivo que lo era *isEnemy* en la **función de pertenencia**: si su valor es distinto a *none*, sabremos que Mario se encuentra en la situación *Mario tiene una moneda o champiñón que da vida cerca*.

[1] En el caso de que haya enemigos en dos regiones distintas, adquirirá el valor de la primera región en la que se encontró un enemigo. El orden de prioridad es *front*, *behind* y *up*.

[2] En el caso de que haya monedas y/o champiñones en dos regiones distintas, adquirirá el valor de la primera región en la que se encontró un enemigo. El orden de prioridad es *front*, *behind* y *up*.

  - En el caso de la **función de evaluación**, podemos saber en qué región del mapa se encuentra ese *Intermediate reward* seguro: delante, detrás o arriba; lo que ayudará a Mario a saber qué acción sería la más adecuada.

- ***isBlocked***. A diferencia de los dos casos anteriores, que servían para orientar a Mario a encontrar posibles fuentes de *Intermediate reward*, lo que presentemos a partir de ahora tendrán la función de que Mario pueda avanzar por el mapa. Para ello, las únicas acciones que nos interesan son moverse a la derecha y saltar; ya que moverse hacia atrás, agacharse o quedarse quieto son contraproducentes en esta situación. Así, la única región de interés pasará a ser la que está justo delante de Mario. De este modo, este atributo booleano tomará en valor *true* si existe algún tipo de obstáculo delante de Mario que no le permite avanzar y *false* en cualquier otro caso. Este atributo nos ayudará a encontrar aquellas situaciones en las que Mario se encuentra bloqueado, es decir, *Mario tiene un obstáculo delante*, de aquellas en las que no (*Mario no tiene un obstáculo delante*). Por tanto, no sirve para comparar instancias en estos dos tipos, ya que en todas ellas esta variable tendría el mismo valor (*true* o *false*). Para ello, utilizaremos los atributos *isOnGround* y *jumped*, que manejan el mecanismo del salto.

- ***isOnGround***. Este atributo booleano tomará el valor *true* si Mario se encuentra pisando el suelo en ese momento y *false* si se encuentra en el aire. Cobra sentido en la función de similitud junto con *jumped*.

- ***jumped***. Se trata también de un atributo booleano: nos indica si Mario saltó en el tick anterior (*true*), o no (*false*). Es un atributo esencial para que Mario no se quede bloqueado, ya que no se puede saltar en el tick actual si en el anterior lo hizo y está actualmente en el suelo. Por tanto, esta variable junto con la anterior dará al agente la información necesaria para saber cómo debe esquivar el obstáculo para la situación *Mario tiene un obstáculo delante*. A pesar de que pensamos en *isOnGround* y *jumped* para sortear los obstáculos, estos dos atributos son los más transversales, y son aplicables a todas las situaciones que definimos.

Los atributos que hemos considerado para la **función de evaluación** son los siguientes:

- ***distancePassedPhys***. El objetivo más sencillo de evaluar es cuánto ha avanzado Mario durante el período de evaluación. Para ello, necesitamos un único atributo que calcule cuánta distancia ha recorrido Mario con respecto al eje x. Teníamos dos opciones: *distancePassedPhys* y *distancePassedCells*. La primera tiene una medida mucho más precisa que la segunda (píxeles y celdas, respectivamente), por lo que decidimos usar aquella que nos proporcionara más información para hacer una evaluación más precisa.

- Para **evaluar la *intermediate reward***, pensamos que no sería buena idea usar directamente el atributo *intermediate reward*, ya que no obtendríamos así qué ha propiciado el aumento o descenso de esa variable. Por ello, vamos a usar la variación de cada uno de los elementos que influyen en la *intermediate reward* de las situaciones que hemos definido para la función de evaluación:

  - ***marioStatus***. Cuando consideramos que Mario podía tener enemigos cerca, una de las posibilidades que existía era que pudieran reducirle la vida al herirle. Hay que tener en cuenta de que a pesar de que Mario también pueda ganar vida recogiendo un champiñón, esto se contabiliza para el cálculo de la *intermediate reward* como que ha obtenido ese nuevo objeto, pero no como que ha ganado vida. De este modo, este atributo numérico nos servirá para saber si Mario ha perdido o ganado vida (con un valor negativo y positivo, respectivamente) y en qué magnitud (1 ó 2). Por tanto, los valores posibles son -2, -1, 0, 1 y 2.

- ***coinsGained***. A partir de este atributo estamos considerando elementos que hacen que aumente la *intermediate reward*. Este primero son las monedas ganadas durante el perído de evaluación, por lo que es un atributo cuyos valores son números enteros y positivos. Hay que tener en cuenta que hemos dividido las monedas y los champiñones para la evaluación, a diferencia de cómo definimos la situación *Mario tiene una moneda o champiñón que da vida cerca* para la función de pertenencia, ya que ambos proporcionan refuerzos positivos diferentes a Mario.
- ***mushroomsDevoured***. Con este atributo obtendremos cuántos champiñones obtuvo Mario durante el perído de evaluación; es decir, un atributo con valores formados por números enteros y positivos, como en el caso anterior.
- ***killsByStomp***. Finalmente, usaremos este último atributo para obtener el número de enemigos que ha eliminado Mario durante el período de evaluación, por tanto, los valores que puede tomar son números enteros y positivos. Se trata de un atributo ciertamente complicado, ya que Mario puede matar enemigos por casualidad, pero tendremos cuidado de que las instancias de entrenamiento sólo contengan situaciones en las que Mario mate a los enemigos cuando los tenga justo delante o detrás (si el salto es muy alto, se pueden superar los *n+12* o *n+24* ticks que tenemos para la evaluación). Por esta misma razón no se incluye la posibilidad de matar a los enemigos con bolas de fuego: el tiempo en el que eliminan a los enemigos es completamente indeterminado.

Por último, guardaremos la **acción que ejecuta Mario** en el tick en el que se recogió la instancia, ya que es necesaria para decirle al bot creado mediante *Instance Based Learning* qué acción debe ejecutar tras elegir la instancia más adecuada.

Llegados a este momento, nos debemos plantear qué agente va a recoger las instancias para crear la base de conocimiento. La resupuesta aquí es muy sencilla: **ninguno de los bot que hemos creado hasta ahora se mueven para maximizar la *intermediate reward***, sino simplemente para avanzar lo máximo posible en el nivel. Si utilizáramos cualquiera de esos bot, Mario nunca tendría en cuenta que la *intermediate reward* es importante, por lo que los bots no son válidos. Por tanto, **las instancias de entrenamiento deben ser recogidas en exclusiva por el humano**. Como las instancias van a ser recogidas por el humano, el abanico de movimientos que el agente puede ejecutar crece con respecto al bot, ya que **el humano tiene libertad para ejecutar todas las acciones que ofrece el juego**. Estas son:

- **Moverse hacia la derecha**.
- **Moverse hacia la izquierda**.
- **Saltar**.
- **Agacharse**.

Todas son útiles para los propósitos que se buscan en esta práctica, **a excepción de agacharse**. De hecho, probamos si esta última acción hacía que bajaras más rápido al suelo tras saltar, pero no es el caso. Teniendo en cuenta que no hay enemigos que lancen objetos que puedan herir a Mario en el nivel más bajo (que es en el que lo vamos a evaluar), no tiene sentido tener incluir esta acción, ya que no nos va a servir para evitar nada.

Por tanto, consideraremos todas las acciones básicas y aquellas que consideren combinaciones de acciones posibles al igual que no ejecutar ninguna de las anteriores (quedarse quieto):

- **Moverse hacia la derecha**, *move_right*.
- **Moverse hacia la izquierda**, *move_left*.

- **Saltar**, *jump*.
- **Saltar mientras se mueve hacia la derecha**, *jump_right*.
- **Saltar mientras se mueve hacia la izquierda**, *jump_left*.
- **Quedarse quieto**, *still*.

Cálculo del número de casillas que significan *cerca*.

Tras definir los atributos para las funciones de similitud y evaluación, recogimos las instancias pertenecientes a la pantalla por defecto con el bot ***P2BotAgent_12ticks_test***. Este bot es el bot procedural que creamos en el *Tutorial 1* modificado para balancear las instancias en las distintas situaciones que explicamos previamente y para guardar instancias con todos los atributos que hemos definido (a excepción de *evaluation*, que no estaba definido por ese entonces). Además, usando el bot en lugar del humano podremos evaluar **cuánto es capaz de moverse Mario en 12 ticks**.[3]

Para ello, usamos el atributo ***distancePassedPhys***, que nos indica la distancia recorrida por Mario en esos 12 ticks para cada instancia con el número de píxeles como unidad. Esto se puede calcular con una resta que calcule la diferencia entre la posición de Mario en *n+12* y en *n* en el eje x (horizontal).

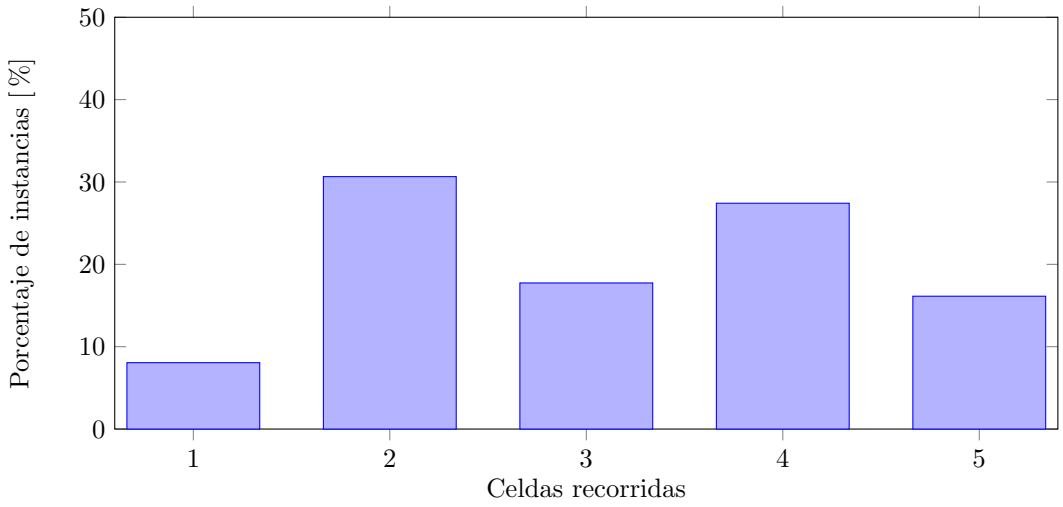

Figura 1: Distribución de las celdas que avanza Mario para *n+12* ticks

Como podemos observar en el fichero en el que se recogen estas instancias, ***calculo_cerca_12_ticks***, **el valor más alto es 66**, que se repite hasta tres veces. Además, vemos que conseguir una distancia mayor de 65 píxeles (4 celdas) no es muy infrecuente: la encontramos en 10 de 62 instancias recogidas, un 16,13 %; como se puede observar en la gráfica.

[3]De las dos posibilidades de ventanas temporales, *n+12* y *n+24*, consideramos que es mejor empezar por ventanas temporales más pequeñas para tener más controladas las acciones de Mario.

Sabiendo que una celda ocupa 16 píxeles, podemos ver que **Mario puede recorrer hasta un máximo de 4,125 celdas en medio segundo**. Teniendo en cuenta que la distancia que estimamos en un principio fue de 2 celdas no es de extrañar que haya instancias en las que Mario no detecte ningún enemigo cerca pero que luego veamos en los atributos de evaluación que eliminó a uno, por ejemplo. Con esta información podemos concluir que **2 celdas alrededor de Mario no son suficientes** para conocer la proporción del mapa a la que puede acceder en los siguientes 12 ticks. Por tanto, la configuración inicial de las variables *isEnemy* e *isCoinMushroom* es incorrecta.

Debido a la inesperada cantidad de celdas que hay que manejar sólo con *n+12* ticks, no vamos a considerar hacer esta práctica con una ventana temporal de *n+24* ticks.

Tras esto, quisimos ver cómo se comportaba Mario con una ventana temporal bastante más pequeña: **6 ticks**. Para ello utilizamos el fichero ***P2BotAgent_6ticks_test***. Este fichero es el mismo que el anterior a excepción de con cuántos ticks del futuro hace la evaluación.

Los resultados que obtenemos en los datos de entrenamiento, recogidos bajo las mismas condiciones que el *.arff* anterior y guardados en ***calculo_cerca_6ticks***, son bastante lógicos: tras reducir hasta la mitad la ventana temporal (un cuarto de segundo), la **máxima distancia recorrida** también termina siendo la mitad: **33 píxeles**. Por tanto, si una celda son 16 píxeles, Mario puede recorrer como máximo **2,0625 celdas**, un número de celdas es mucho más manejable que el anterior.

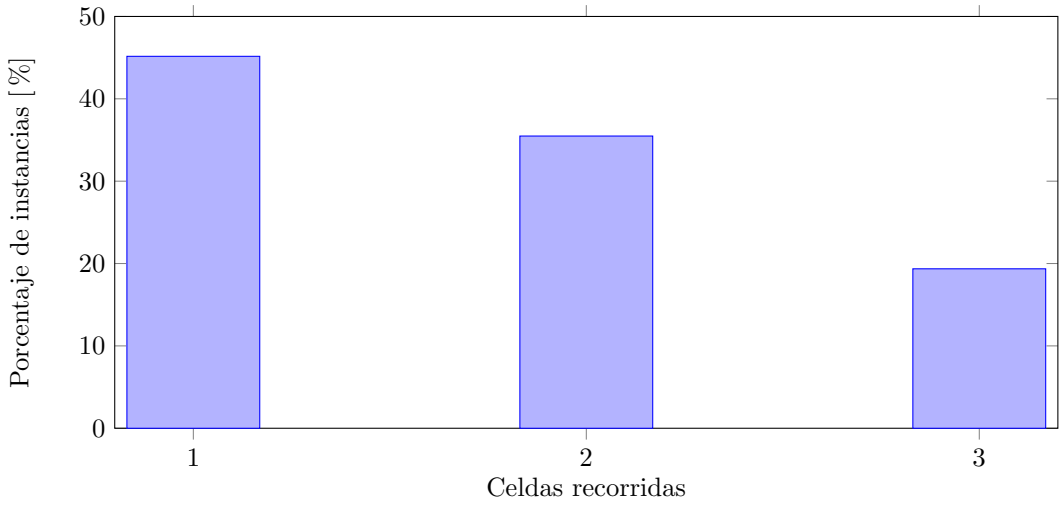

Figura 2: Distribución de las celdas que avanza Mario para *n+6* ticks

A pesar de que Mario puede avanzar un píxel más de lo que constituyen dos celdas con relativa facilidad, como se puede apreciar en la gráfica (12 de 62 de las instancias, lo que da un resultado del 19,36 %), es una medida tan pequeña puede considerarse despreciable. De hecho, **considerar 3 celdas para el entorno de Mario sólo por un píxel, sería añadir ruido de forma innecesaria**. De este modo, si elegimos una distancia temporal de *n+6* ticks, **lo correcto sería** mirar el espacio que rodea a Mario con **hasta 2 celdas de distancia de él**, tal y como está ahora.

Así, sólo **nos queda comprobar** que **si** la distancia temporal es de *n+6* ticks y los atributos *isEnemy* e *isMushroomCoin* comprueban si hay enemigos y monedas / champiñones con hasta 2 celdas de distancia, **las instancias son coherentes** (es decir, que no salga que Mario ha acabado con un enemigo en los atributos de evaluación si en su área de visión para los siguientes 6 ticks no se encontraba ningún enemigo, por ejemplo). Si conseguimos esto, podemos continuar con el aprendizaje de Mario. Comprobamos cada una de las **situaciones que pueden poner en riesgo esta coherencia**:

- **Mario no tiene un enemigo *cerca***, pero en los atributos de evaluación aparece que **ha matado al menos a un enemigo**. Encontramos este caso en la instancia de la **línea 42**, para una situación en la que Mario estaba bloqueado. Viendo que saltó en esa instancia (*jump_right*), es muy probable que tras saltar el obstáculo, llegara a una nueva plataforma que tenía un enemigo en el borde, por lo que lo eliminó. Se trata por tanto de un caso muy particular que no consideramos.
- **Mario no tiene una moneda / champiñón *cerca***, pero en los atributos de evaluación aparece que **ha conseguido al menos una moneda o un champiñón**. Encontramos este caso en la instancia de la **línea 43**, para una situación en la que Mario tenía un enemigo delante, pero no monedas *cerca*. Al igual que en la situación anterior, es muy probable que tras saltar para evitar al enemigo (*jump_right*), llegara a una nueva plataforma en la que había monedas.

Como podemos ver, se dan esas dos situaciones en dos ocasiones, por lo que son casos puntuales, *outlayers*. Por tanto, nuestro fichero de instancias ***calculo_cerca_6ticks* se puede considerar coherente**.

Una vez que hemos comprobado que las celdas que tienen en cuenta *isEnemy* e *isCoinMushroom* son las adecuadas, **explicaremos en detalle a qué nos referimos con *2 celdas de distancia a Mario*** usando el siguiente dibujo:

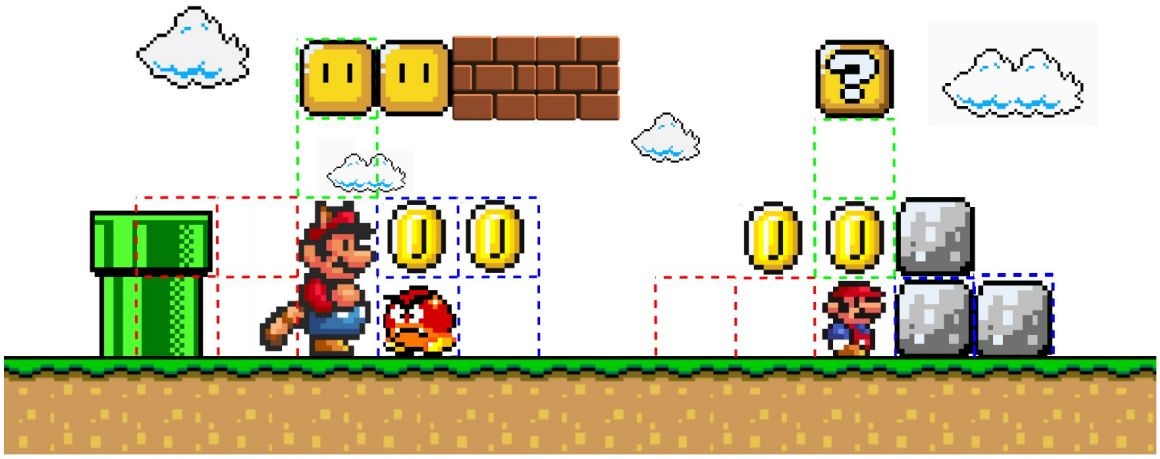

Figura 3: Matriz de observación tenida en cuenta para la recogida de datos.

En la imagen se representan las regiones anteriormente descritas haciendo uso de colores. Estos colores se corresponden de la siguiente forma:

- *front*
- *behind*
- *up*

Como se puede apreciar, observamos un **tamaño distinto del mundo de acuerdo al estado en el que se halla Mario**; de tal manera que se siguen observando las celdas delanteras, traseras y de arriba, pero de acuerdo a su altura. La decisión de ajustar las celdas de la matriz de observación fue tomada al darnos cuenta que con **una reducción de altura Mario se expone a menos peligros**, ya que el área en la que puede ser dañado se ve reducida. Además, el alcance de su salto también se ve disminuido al pasar a ser pequeño.

Por último, decidimos **no tener en cuenta las celdas diagonales** debido a que serán examinadas según Mario avance con las celdas de la región de *up*. Consideramos que es entonces cuando debemos plantearnos si saltar para recogerla o no.

4. Creación de la base de conocimiento

Para crear la base de conocimiento que utilizará luego Mario para la fase de *Instance Based Learning*, lo primero que nos planteamos es **qué agente debía ser el responsable de la captura de datos**. Este asunto lo resolvimos anteriormente diciendo que ninguno de los bot que habíamos creado hasta la fecha se movía para maximizar la *intermediate reward*, además de que sus acciones están limitadas a dos movimientos (*move_right* y *move_jump*). Estos dos motivos nos llevan a pensar que **el agente humano es el único que será realmente útil para para el aprendizaje de Mario**.

A pesar de que las instancias del bot no son útiles a priori, **recogeremos instancias tanto con el humano como con el bot**, para ver cómo afecta al comportamiento de Mario, es decir, ***P2HumanAgent*** y ***P2BotAgent***, respectivamente. Para las instancias que recojamos con el humano, vamos a marcar primero el **comportamiento** que le vamos a dar durante las partidas, pensado a partir de cómo definimos nuestros atributos. Jugaremos siempre con la G activada, para saber qué es lo que puede ver Mario en cada momento.

Como norma general, **Mario se debe mover siempre hacia la derecha**, teniendo en cuenta los siguientes eventos:

- Si se encuentra con algún tipo de **obstáculo que lo bloquea**, debe sortearlo lo más rápidamente posible.
- Si se encuentra con algún **champiñón o moneda** y está dentro de su rango de visión, debe ir a recogerlo.
- Mario **debe intentar eliminar a todos los enemigos que entren en su rango de visión**, pero siempre saltando cuando esté justo delante o detrás de ellos. No debemos esperar a los enemigos bajo ninguna circunstancia.
- Si alguna de las situaciones anteriores **pone en peligro la vida de Mario** y es muy posible que se pierda vida, **se debe evitar a toda costa**.

Sabiendo que la elección de la acción se debe hacer el tiempo real, la base de conocimiento no debe ser excesivamente grande. En las anteriores pruebas con *P2BotAgent_12ticks_test* y *P2BotAgent_6ticks_test*, vimos que **se recogían unas 62 instancias por partida**. Tomando ese valor como referencia, con unas 20 partidas obtendríamos unas 1240 instancias. Idealmente, todas las situaciones tendrán el mismo número de instancias, por lo que serían **unas 310 para cada una**. Esas 20 partidas **se jugarán en semillas diferentes**, de la 0 a la 19; aunque todas en la dificultad inicial, ya que no se requiere que Mario aprenda a jugar en distintas dificultades, sino en todo tipo de escenarios.[4]

Tras recoger las instancias, **vimos cómo se habían repartido las acciones en las diferentes situaciones**, tanto para el bot como para el humano. El **bot** tras 20 partidas ha recogido **495 instancias**, que son muchas menos de las que esperábamos. Esto se debe a que Mario se queda bloqueado en muchos de los niveles que probamos. Los resultados del bot se encuentran en ***base_conocimiento_bot*** y son los siguientes:

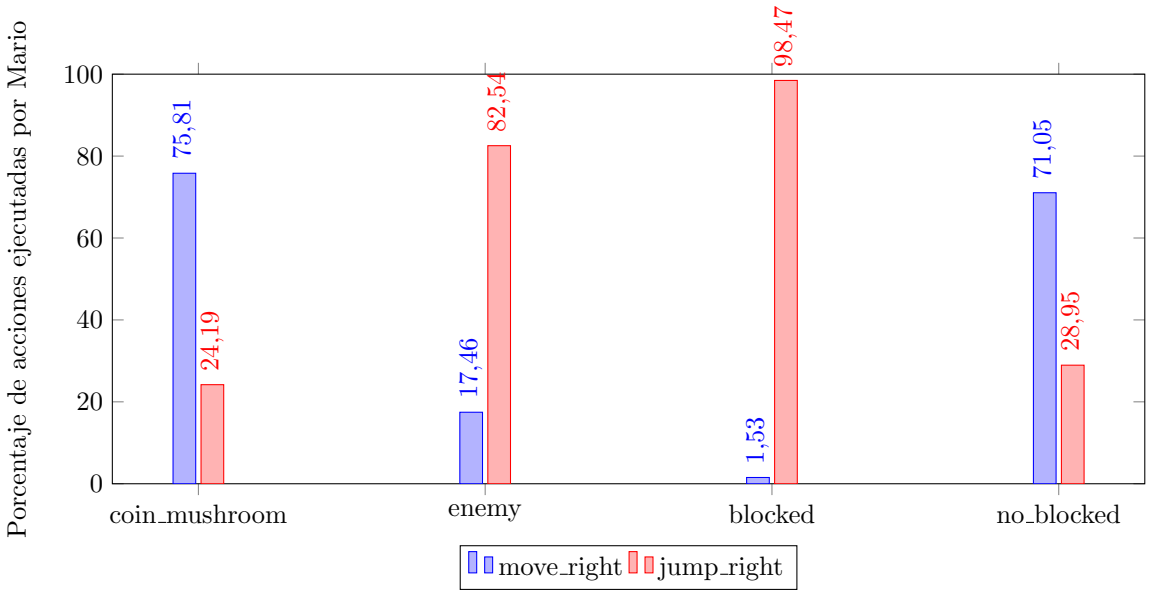

Figura 4: Reparto de acciones en las distintas situaciones para el **bot**

Por un lado, **el balanceo de instancias se ha conseguido correctamente**, y las cuatro situaciones tienen un número de instancias muy similar:

- 124 instancias para ***coin_mushroom*** (25,05 %).
- 126 instancias para ***enemy*** (25,46 %).
- 131 instancias para ***is_blocked*** (26,47 %).
- 114 instancias para ***no_blocked*** (23,03 %).

Por otro lado, la proporción que se muestra en la figura de las acciones tiene bastante sentido:

- Como Mario no busca maximizar la *intermediate reward*, el movimiento mayoritario en ***coin_mushroom*** para el bot moverse a la derecha. En realidad, lo que le ha motivado a saltar en estas situaciones son obstáculos y enemigos que se encontró por el camino, lo que no es incompatible con que haya monedas o champiñones cerca.

[4] Aplicaremos este método tanto para recoger instancias con el bot como con el humano. En el caso del bot, esto se lleva a cabo con el script *recog_instancias*.

- El caso de ***enemy*** es totalmente opuesto: Mario salta siempre que puede cuando ve un enemigo. Que haya instancias en esta situación en la que Mario no salte es porque él sólo considera los enemigos que tiene enfrente, cuando nuestra variable *isEnemy* se consideran enemigos tanto delante, como detrás y arriba.
- La situación de ***blocked*** es mucho más radical, ya que *isBlocked* sí que considera los obstáculos sólo delante, al igual que el bot. Por tanto, es lógico que la acción de saltar roce el 100 %.
- Finalmente, la situación de ***no_blocked*** ofrece un abanico muy amplio de posibilidades, ya que es la situación más abierta. Así, en una situación de no estar bloqueado, Mario se moverá en más ocasiones a la derecha que saltará, ya que la única razón para saltar sin estar bloqueado son los enemigos, cuya aparición no es tan frecuente.

Una vez vistos los resultados del bot, vamos a analizar los **resultados** que nos proporciona ***base_conocimiento_human***. En un primer lugar, el **número de instancias recogidas** es bastante mayor que con el bot (**817**), aunque siguen siendo muchas menos de las que previmos. Suponemos que los datos de la estimación no eran los adecuados y en el ejemplo que tomamos como referencia se obtuvieron un número excepcional de instancias. Los datos del humano son los siguientes:

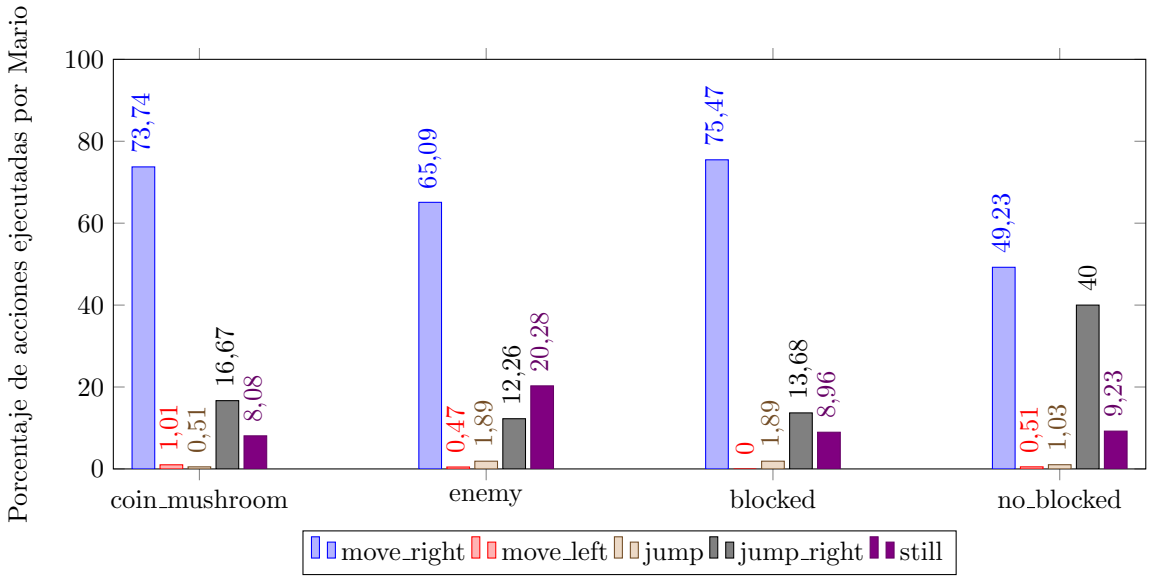

Figura 5: Reparto de acciones en las distintas situaciones para el **humano**

Observando los datos obtenidos del entrenamiento con el humano, podemos observar un **gran problema en la forma en la que se han recogido las instancias, tanto para la situación *enemy* y *blocked***:

- Para el caso de la situación ***enemy***, observamos que en un 65,09 % de instancias, Mario se mueve hacia a la derecha. Comparando con los resultados del bot, que saltó un 82,54 % de las veces, podemos ver que algo está yendo mal. Tras un largo período de pruebas y de observar cómo se generaban las instancias, pudimos comprobar la causa de todo ello, que no es más que la forma en que se recogen las instancias. Para empezar, esta situación se da cuando Mario detecta un enemigo, que es con dos celdas de antelación. Teniendo en cuenta este factor y el hecho de que es imposible que un jugador humano detecte en todos los casos cuándo está a esas dos celdas del enemigo, es lógico que en la mayoría de los casos se esté moviendo hacia la derecha, ya que puede saltar a cualquier otra distancia para evitarlo. Eso sumado a que no cogemos otra instancia de la misma situación hasta que han sucedido las otras tres de esa

ronda, hace bastante complicado que se registre el tick en el que Mario ha saltado para evitar a ese enemigo. Además, otro efecto contraproducente es que estas instancias suelen recibir una evaluación positiva si el *goomba* no ha herido a Mario, debido al salto que se hizo poco después y cuyos efectos han podido ser evaluados dentro de los *n+6* ticks.

- El caso de la situación ***blocked*** es similar: en un 75,47 % de las instancias Mario se está moviendo hacia la derecha en lugar de saltar. Para haber obtenido unos resultados más óptimos, Mario debería haber saltado con regularidad tras llegar a una celda de distancia de la que se encuentra el obstáculo. Sin embargo, las instancias que ha recogido en su mayoría son aquellas en las que el agente humano se acercaba al obstáculo para saltarlo. En estas situaciones observamos también recompensas altas, ya que evalúa lo que consiguió Mario tras saltar dicho obstáculo, a pesar de que la acción no se vea reflejada.

Tras darnos cuenta de estos problemas y ver en algunas pruebas con el agente final, ***P2BotAgent_IBL***, que tenía serios problemas para avanzar por el escenario, **empezamos a tratar los datos de la base de conocimiento**. El primer problema que detectamos es que **Mario tenía demasiadas instancias con la acción *still***. Esto hacía que se quedara parado en muchas ocasiones en lugar de saltar los obstáculos u enemigos. Decidimos que Mario debería estar siempre en movimiento para evitar que se quedara quieto en un punto del mapa, por lo que **eliminamos todas las instancias que contenían esta acción**. Guardamos el resultado en *base_conocimiento_human_noSTILL*. Tras esto, la base de conocimiento resultó así:

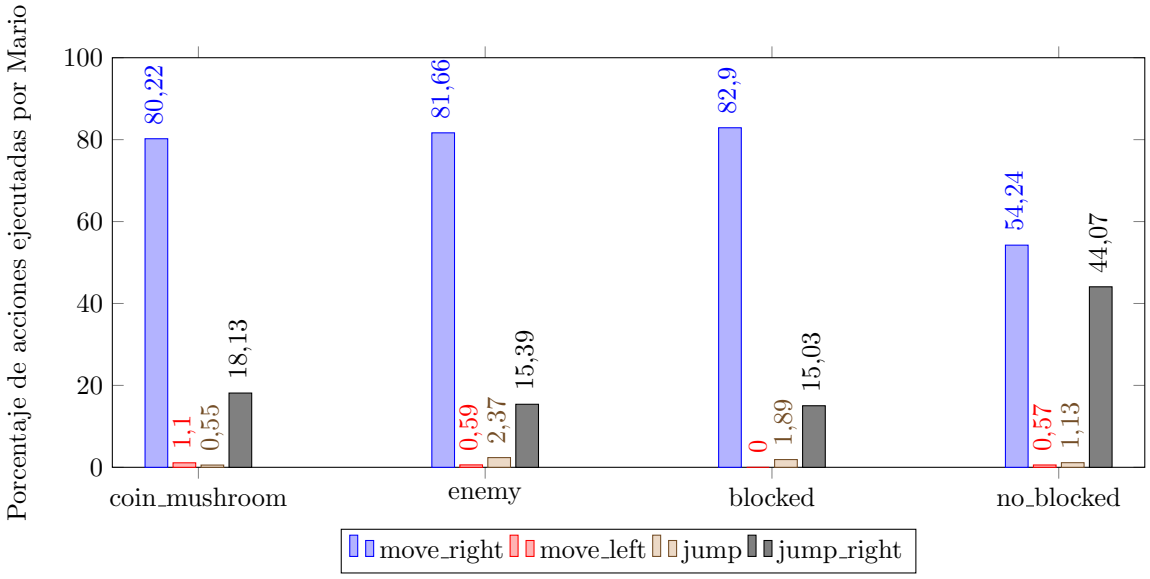

Figura 6: Reparto de acciones en las distintas situaciones tras quitar las instancias con la acción *still*

Observando los resultados, podemos ver que se acentúa la diferencia entre *move_right* y *jump_right*, lo que va a suponer **un gran problema a la hora de poder saltar los obstáculos y los enemigos**, ya que existen muy pocas instancias en esas situaciones en las que Mario salte. Tras comprobar que no salta los enemigos ni los obstáculos, probamos a **eliminar la mitad de las instancias que contienen *move_jump* como acción para la situación *enemy***. Guardamos estos resultados en *base_conocimiento_human_fixedBlocked* y procedemos a observar el nuevo reparto de acciones:

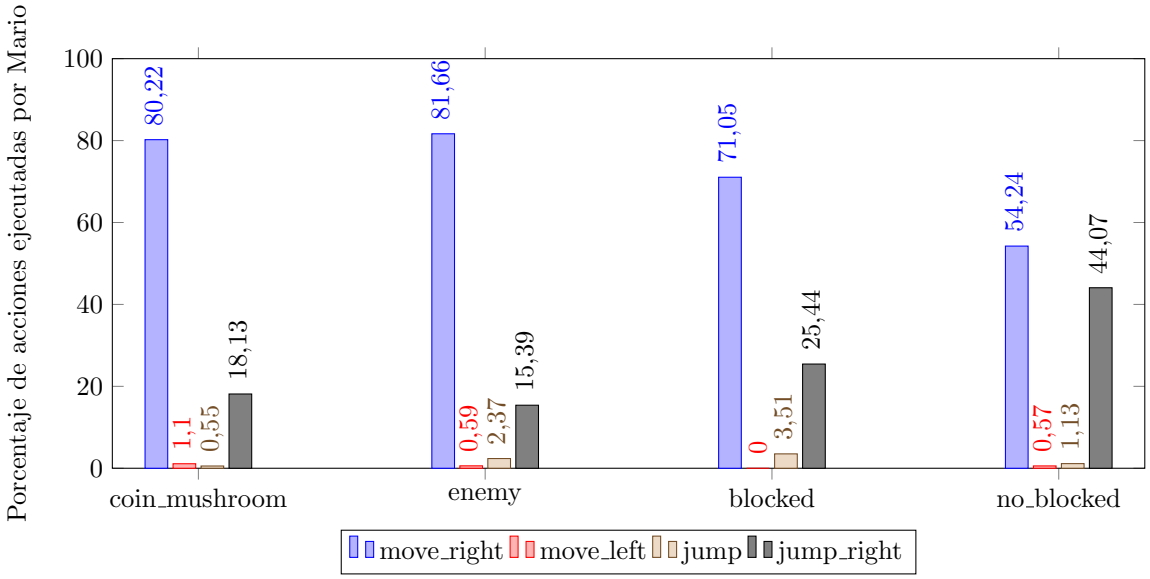

Figura 7: Reparto de acciones en las distintas situaciones tras el ajuste en *blocked*

Como podemos observar, el número de instancias que contienen la acción *move_jump* se han reducido considerablemente, por lo que su diferencia con *jump_right* no es tan exagerada como antes. De este modo podemos ver que **Mario**, esta vez sí, **consigue saltar bastantes obstáculos, aunque no sin cierta dificultad** (tarda un poco en saltarlos, ya que se suele quedar un rato moviéndose hacia delante delante del obstáculo).

El siguiente problema que observamos es que **Mario no suele saltar cuando ve a los enemigos** porque tiene demasiadas instancias con la acción *move_right* para esta situación, muchas incluso con evaluaciones altas. Por ello, decidimos eliminar esta vez no tanto en cantidad, sino en calidad: **quitamos todas aquellas instancias de la situación *enemy* que contengan la acción *move_right* y una evaluación mayor de 40 puntos**. De esta forma, las instancias con *jump_right* tendrán más posibilidades de ser las elegidas. Los resultados se encuentran en *base_conocimiento_human_fixedEnemy* y proveen los siguientes resultados:

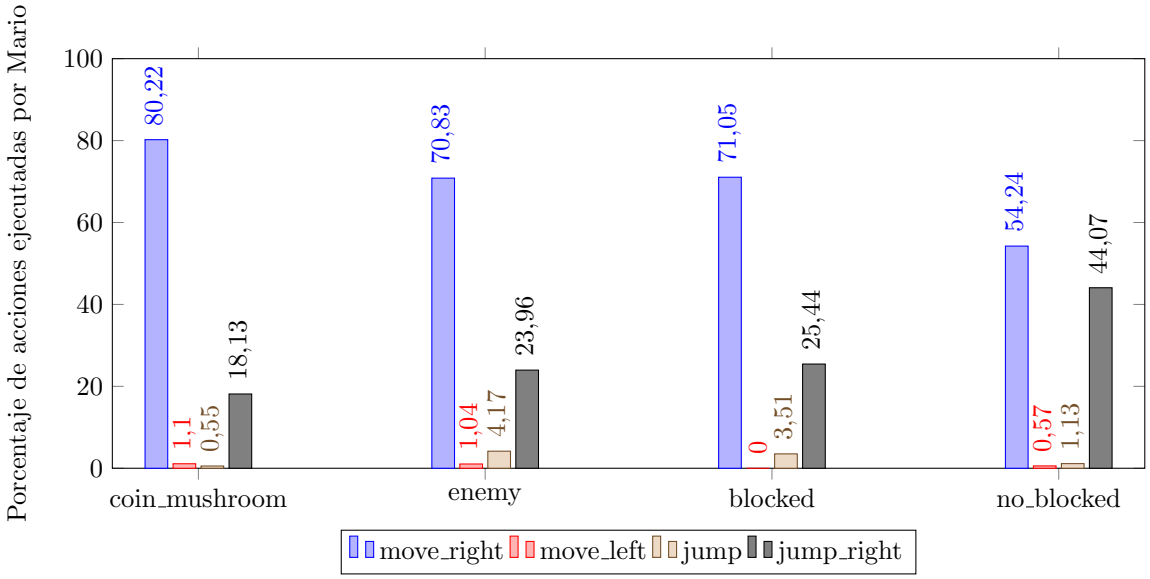

Figura 8: Reparto de acciones en las distintas situaciones tras el ajuste en *enemy*

Con este ajuste de la base de conocimiento conseguimos una mejora similar que en el ajuste anterior, siendo las proporciones de *move_right* y *jump_right* bastante parecidas, de hecho. Comprobamos entonces qué resultados consigue en el comportamiento de Mario. Tras observar unas cuantas partidas, podemos ver que **salta a los enemigos bastante mejor que lo hace con los obstáculos, no sin pasar por encima de alguno perdiendo vida**.

Como vemos que Mario sigue teniendo más dificultad en saltar a los bloques que a los enemigos, probamos un último ajuste con la situación *blocked*: **eliminamos las instancias de una forma análoga que en el caso anterior, aunque quitando aquellas con una evaluación mayor de 30**, ya que las instancias de esta situación tienen unos resultados de evaluación mucho más bajos que en *enemy*. Estos resultados se guardan en *base_conocimiento_human_fixedBlocked_more*. La proporción que obtenemos de las acciones es la siguiente:

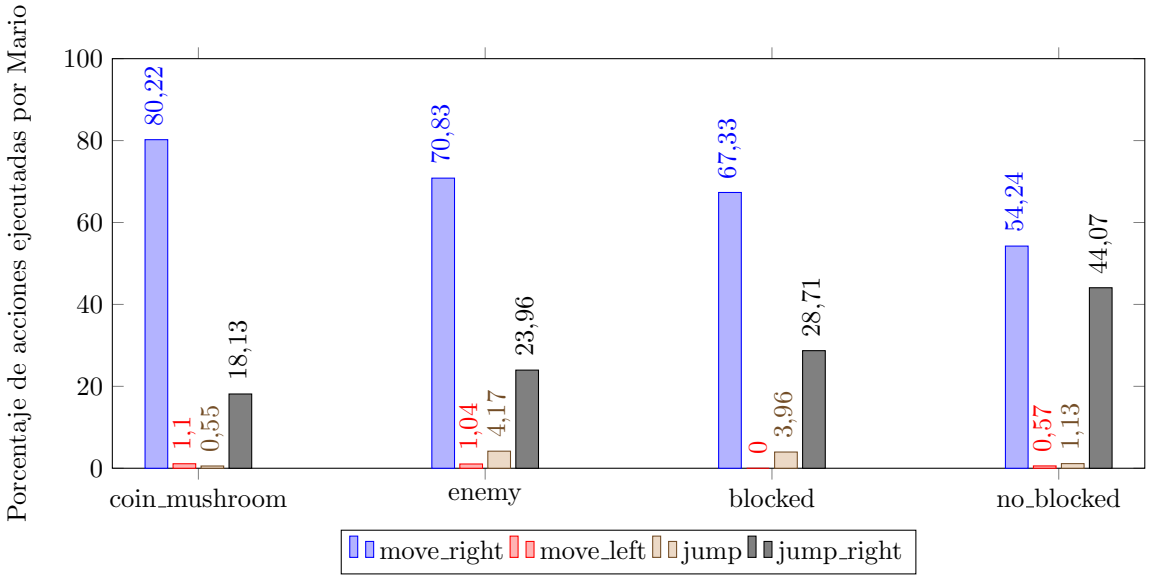

Figura 9: Reparto de acciones en las distintas situaciones tras el ajuste final de *blocked*

Después de este último ajuste, vemos que Mario ha mejorado un poco con su salto; aunque sigue dudando a la hora de saltar los obstáculos. A pesar de ello, **consideramos que su comportamiento es aceptable tal y como está**; aun sabiendo que no hemos solucionado la raíz del problema: el balanceo de instancias. Habríamos tratado de arreglarlo mejor si hubiéramos tenido más tiempo para trabajar en esta práctica, pero nos ha sido imposible.

5. Tratamiento sobre los datos

El principal tratamiento que vamos a hacer sobre los datos es el del **balanceo de instancias** presentado anteriormente. Se trata de un proceso basado en contadores: cada situación tiene un contador asociado (*near_coin_mushroom_counter*, *near_enemy_counter*, *is_blocked_counter*, *no_blocked_counter*). Con este proceso pretendemos **grabar en la base de conocimiento el mismo número de instancias de cada una de las situaciones que hemos identificado**. El proceso es el siguiente:

1. El contador *no_blocked_counter* se inicializa a 1 en el primer tick, ya que es la situación más común de todas.
2. Comprobamos si la instancia actual pertenece a una situación u otra, dándole un orden de precedencia (una misma instancia puede pertenecer a varias situaciones distintas). Usamos la variable *visited* para comprobar que no asignamos una instancia a más de una situación.
3. Si la instancia corresponde a una cierta situación distinta de *no_blocked_counter*, comprobamos que el contador de esa situación es menor que el de la situación base (*no_blocked_counter*). Con esto nos aseguramos que guardamos el mismo número de instancias de cada situación.
    - Si es menor que dicho contador, escribimos la instancia completa, añadiéndole el atributo de *situation*, que era desconocido hasta la fecha. En este momento, *visited* pasaría a ser a ser *true*, por lo que esta instancia no se identificaría como ninguna de las otras situaciones.
    - En el caso contrario, seguiremos mirando en cada una de las situaciones restantes, ya que esta instancia podría pertenecer a otra situación cuyo contador es menor que *no_blocked_counter*.

4. Si todavía la instancia tiene opciones de ser del tipo *no_blocked_counter*, comprobaremos si todos los contadores se han igualado ya al de esta situación. Sólo en ese momento escribiremos esta instancia y aumentaremos su contador, dando la opción otra vez de que podamos grabar más. Este sería nuestro principal método de control de la cantidad de instancias que se escriben de cada situación, *por rondas*.

Además de la pre-indexación de las situaciones que se ejecuta durante la recogida de datos, llevamos a cabo una **selección de las instancias más adecuadas** una vez que recogimos todas las que necesitábamos. El propósito de este ajuste fue intentar mejorar el comportamiento de Mario, debido a que nuestro balanceo de instancias no recogía bien los datos por las particularidades del agente que utilizamos, el humano. Todo este proceso lo vimos en detalle en la anterior sección.

6. Estructuras de datos utilizadas

Vamos a utilizar principalmente **dos estructuras de datos**, siendo cada una de ellas útil para cada una de las partes de las que consta esta práctica: recogida de instancias de la base de conocimiento e *Instance Based Learning*.

6.1. Fase de recogida de instancias de la base de conocimiento

Se necesita una estructura de datos para **guardar las instancias que todavía no han sido tenidas en cuenta para ser grabadas en la base de conocimiento**. Esto se debe a que todos los atributos de evaluación, a excepción de la acción, se definen a través de una resta de la característica que nos interesa de Mario para ese atributo para *n+6* y *n*.

Por tanto, una vez obtenida toda la información que podemos recolectar en el tick presente para la instancia actual, **necesitamos guardarla para luego poder recuperarla dentro de 6 ticks**. La forma en que hemos guardado estas instancias es creando una ***linked list* de la clase *P2Element***, ya que, a pesar de tener siempre un tamaño fijo, como veremos más adelante, necesitamos poder añadir y eliminar sus elementos dinámicamente, lo que en un array sería engorroso. Esta clase contiene los atributos *id* y *line*: el primero guarda el tick en que el tuvo lugar la instancia, y el segundo es un array que guarda como *String* cada uno de los argumentos de la instancia.

Esta *linked list* sólo contiene **6 elementos al mismo tiempo**, ya que al igual que guardamos una instancia cuando ya hemos terminado de obtener sus atributos, esta desaparece de la lista cuando ha sido analizada para ser grabada en el fichero (o no). Si mantuviéramos las instancias permanentemente en la lista, sería un gasto innecesario de la memoria. A pesar de que se trata de una lista, sólo se trabaja con ella con el primer y último elemento, **como si fuera una *queue***:

- Guardamos la instancia en la última posición tras obtener todos sus atributos con ***add()***.
- Obtenemos la instancia más reciente, la última, para escribir qué acción ha ejecutado Mario en el tick actual para el atributo *action* usando ***getLast()***. (Este atributo se recoge por separado porque se encuentra en un método distinto al resto.)
- Obtenemos la instancia más antigua, la primera, para sobreescribir sus datos de evaluación usando ***getFirst()***.
- Extraemos la instancia más antigua, la primera, cuando tenemos intención de escribirla en la base de conocimiento con ***poll()***.

6.2. Fase de *Instance Based Learning*

Para este caso es necesaria una **estructura de datos para crear nuestra base de conocimiento a partir de las instancias recogidas durante la fase anterior**. Esta debe ser accesible durante el proceso de decisión de acciones en tiempo real, ya que es esencial para las tres funciones que componen el *Instance Based Learning*.

Pensamos en crear una **estructura diferenciada para cada una de las cuatro situaciones que se pueden dar**, teniendo en cuenta que la función de similitud compara todas aquellas instancias que pertenecen a la misma situación. Esas cuatro estructuras serían exactamente iguales, con la única diferencia de la información que contienen.

Sabiendo que **el número de elementos que debe guardar cada una de las estructuras está determinado por del número de instancias recogidas en la base de conocimiento para cada situación**; podemos utilizar estructuras con longitud fijas, como los **arrays**. Esta opción es mejor que una *linked list* en este caso concreto porque, a menos que trabajemos sólo con el primer y último elemento de la lista o que esta deba crecer dinámicamente, una *linked list* será siempre más ineficiente que un array. Para este caso concreto, debemos poder obtener cualquier elemento de la lista independientemente de la posición en la que se encuentre.

Este array está formado por una **estructura que representa una instancia, *instance***, de forma análoga a como se hizo con *P2Element*, sólo que los atributos que necesitamos son diferentes. Estos **contienen exclusivamente la información necesaria para la función de similitud y el resultado de la función de evaluación**, ya que la situación a la que pertenecen es inherente a la estructura en la que se encuentran y no necesitamos los atributos de la función de evaluación, porque esta ya ha sido calculada. Los atributos que contiene son los siguientes: *id*, el identificador de las instancias; los atributos necesarios para la función de similitud (*isEnemy*, *isCoinMushroom*, *isBlocked*, *isOnGround*, *jumped*); la acción que tomó Mario, *action* y, finalmente, la evaluación de dicha instancia, *evaluation*. Todos los atributos son del tipo String a excepción de *evaluation*, que se pasará directamente como *double* para agilizar la función de evaluación (aquí debemos comparar números, pero en la función de similitud sólo necesitamos saber qué valor nominal tiene cada atributo).

Esta estructura de datos **se inicializa con todas las instancias recogidas en la fase anterior haciendo uso del método *init_IBL_database***. Este método es llamado por el constructor, ya que debe estar listo antes de empezar a tomar decisiones en tiempo real. Para ello, el constructor crea un *FileReader* y un *BufferedReader* para poder **leer del fichero que compone la base de conocimiento**; cuyo nombre se puede seleccionar libremente, siempre que lo indiquemos correctamente y que se encuentre en la carpeta adecuada.[5] Tras esto, ignora las 17 primeras líneas del fichero, que son las que componen el *header* que necesita Weka para leer los archivos *.arff*. Por tanto, a partir de la decimoctava línea ya podemos encontrar la información de las instancias. Como hay una instancia por línea, leemos cada una de ellas haciendo uso de *readLine()* hasta que ya no queden más. Es durante ese proceso cuando guardamos la instancia en un array de String, con una posición para cada elemento separado por comas. Este es el argumento que luego pasamos como argumento a *init_IBL_database*.

[5] Nosotros hemos puesto sólo el nombre del archivo, por lo que este se debe encontrar en la misma carpeta que se encuentre *ejecutar.bat* para que funcione.

Ya en *init_IBL_database*, **dependiendo de la situación a la que pertenezca la instancia** (esta información se puede obtener a través de duodécimo atributo, *situation*), **la guardamos en uno de los cuatro arrays que hemos definido para cada una de las situaciones**. Para poder guardar cada instancia en la posición adecuada dentro de cada array, creamos cuatro contadores como variables globales para saber cuál era la siguiente posición del array en la que teníamos que escribir (*read_coin_mushroomInstance*, *read_enemyInstance*, *read_blockedInstance*, *read_no_blockedInstance*). Finalmente, hay que tener en cuenta que no necesitamos todos los atributos que hemos recogido del fichero para esta nueva *instance*, por lo que hacemos uso de sus posiciones relativas para guardar sólo los que nos interesan.

7. Funciones del *Instance Based Learning*

Para aplicar el proceso del *Instance Based Learning* a nuestro agente, aplicaremos tres funciones: la función de pertenencia, la de similitud y la de evaluación.

7.1. Función de pertenencia

El **objetivo** de la función de pertenencia es indicar a **qué situación pertenece la instancia que está analizando**. Estas situaciones son las mismas que se definieron en el apartado del balanceo de instancias.

Esta función se debe aplicar tanto a las instancias pertenecientes a la base de conocimiento como a las que nos llegan durante la fase de *IBL* en tiempo real, ya que estas últimas sólo se deben comparar con aquellas que pertenezcan a su misma situación y estas son las que se encuentran dentro de la base de conocimiento.

Existen **dos opciones** para etiquetar las instancias de la base de conocimiento en una situación u otra:

- La primera, que es la que hemos elegido, es **colocar la situación a la que pertenece como un atributo** mientras recogemos instancias. Así, sólo tendremos que recuperarla cuando la vayamos a guardar en la estructura de datos correspondiente a su situación.
- La segunda, que es **obtener** a partir de los atributos de la instancia **la situación a la que pertenece durante la fase de *IBL***. Después se guardaría igualmente en la estructura de datos correspondiente.

La segunda opción, de llevarse a cabo, no supondría ningún riesgo para la toma de decisiones en tiempo real, ya que se hace en el constructor. A pesar de ello, hemos preferido la primera, ya que **no nos parece buena idea hacer dos veces las mismas comprobaciones lógicas para cada instancia**, siendo estas son inevitables durante el balanceo de instancias.

Sin embargo, **clasificar las instancias que nos llegan durante la fase de *Instance Based Learning* se debe hacer inevitablemente en tiempo real**, ya que no conocemos estas instancias con anterioridad como las otras.

La función de pertenencia es tan sencilla como utilizar los atributos guardados de una instancia para identificar a qué situación pertenece:

- **Mario tiene una moneda o champiñón que da vida *cerca***. Para que se de esta situación basta con que el valor de *isCoinMushroom* sea distinto de *None*.
- **Mario tiene un enemigo *cerca***. Para que se de esta situación basta con que el valor de *isEnemy* sea distinto de *None*.

- **Mario tiene obstáculo delante**. Para que se de esta situación basta con que el valor de *isBlocked* sea *true*.
- **Mario no tiene un obstáculo delante**. Para que se de esta situación basta con que el valor de *isBlocked* sea *false*.

A pesar de que la pertenencia a una situación u otra es exactamente igual para ambos tipos de instancias (tanto para las de la base de conocimiento como para las que recibido en tiempo real), la **forma en la que estas situaciones son asignadas** difiere completamente:

- La forma en que asignamos situaciones a las de la **base de conocimiento**, que luego se pasa como atributo; fue la que definimos en la sección del *Tratamiento sobre los datos*. Se encuentra implementada dentro del método *getTrainingFile()*.
- Para las **instancias que recibimos en tiempo real**, como no hacemos uso de contadores; **hemos asignado las situaciones en función de lo importantes que sean para obtener los objetivos del juego**, ya que una misma instancia puede pertenecer a más de una situación. La prioridad que aplicamos es la siguiente:

  1. Lo que más nos preocupa es que Mario pierda vida. Por tanto, para que se pueda pasar los niveles, nos tenemos que fijar primeramente en si la situación que hemos recibido contiene enemigos; ya que esta es la única fuente de peligro que podemos encontrar. Consecuentemente, la primera situación a considerar es **Mario tiene un enemigo *cerca***.
  2. La siguiente prioridad es que Mario no se quede bloqueado, ya que sin avanzar no conseguiría ninguno de los dos objetivos. Por tanto, la segunda situación que miramos es **Mario tiene un obstáculo delante**.
  3. La única situación que queda por comprobar que afecte a la consecución de alguno de los objetivos es la de **Mario tiene una moneda o champiñón que da vida *cerca***, por lo que es la tercera que miramos.
  4. Finalmente, nos queda la situación en la que Mario no está bloqueado, (y en la que tampoco tendrá enemigos, monedas / champiñones ni obstáculos que no le dejen avanzar, ya que antes ha debido pasar por las otras tres situaciones); que es la que tiene menos interés. La situación que tomará llegados a este caso será **Mario no tiene un obstáculo delante**.

Esta última se encuentra implementada en el método ***func_pertenencia()***, que devuelve el nombre de la situación a la que pertenece la instancia. **Se llaman a todas las funciones que usamos para el *Instance Based Learning* de manera consecutiva dentro del método *getAction()***, ya que el objetivo de estas tres (pertenencia, similitud, evaluación) es obtener la acción que Mario debe realizar.

7.2. Función de similitud

El **objetivo** de la función de la similitud consiste en, recorriéndose el array que corresponde a la situación que pertenece la instancia que hemos obtenido para el tick actual, **obtener las N más parecidas a esta dentro de la situación a la que pertenece**. Por tanto, se trata de una función que debe ser ejecutada exclusivamente durante la etapa de *Instance Based Learning*, ya que necesitamos tanto las instancias de la base de conocimiento como la que obtenemos en tiempo real para poder ejecutarla.

Tenemos que admitir que la función de similitud fue la más difícil de determinar de las tres y que la mayoría de las decisiones que tomamos sobre ella fue probando diferentes valores y viendo cómo reaccionaba Mario a ellas. Su **implementación** completa se puede encontrar dentro de la función ***func_pertenencia()***, un método que devuelve el array *similars*, que contiene las N instancias más similares a la dada. Este es el array que luego usará la función de evaluación para elegir qué acción debe tomar Mario.

Lo primero que tuvimos que elegir fue **cuánto debía valer N**. Desde un principio sabíamos que el valor de N tenía que ser elevado, ya que los atributos que elegimos para la función de similitud no dan mucho juego, debido a que no son muchos y sus valores son nominales. Por esa casuística, **el número de situaciones que podemos definir con ellas no son muchas**, además de que estar en una situación u otra fija algunos valores de algunos atributos (por ejemplo, el atributo *isEnemy* nunca será *None*) y eso reduce las posibilidades. Por tanto, elegimos que **N fuera igual a 10**.

Sin embargo, haber elegido una N grande no significa que siempre haya un número N de instancias más similares a una dada, es decir, **puede haber empates**. Estos empates en la función de similitud se ocasionan cuando la instancia que tiene una *puntuación* más baja de las N instancias más similares hasta la fecha empata con la que estamos evaluando para el tick actual. **El mecanismo que hemos elegido para romper estos empates es la aleatoriedad**, ya que si son igual de similares, no podemos posicionarnos a favor de ninguna de ellas, así que lo decida la suerte.

La implementación de este mecanismo sería muy sencilla: haciendo uso de un ***Math.random()*** en el caso de empate:

- Si la función obtiene un 1, cambiaremos la instancia que había entrado en *similars* anteriormente saldrá para dejar paso a la que generó este empate.
- En el caso contrario, *similars* tendrá las mismas instancias que antes del empate.

Introducir esta aleatoriedad provoca que **Mario varíe su comportamiento en cada jugada**, lo que añade cierto factor sorpresa. Sin embargo, comprobamos que esta aleatoriedad sólo afecta a cuándo salta un bloque y si salta un enemigo o no (aunque la gran mayoría de las veces los salta).

Es importante remarcar también que creamos una **nueva estructura llamada *instance_similitud*** para trabajar sólo con la información que necesitamos manejar para comparar instancias, y no *instance*, como venía siendo habitual hasta ahora. Esta estructura **está formada por *id***, que es la posición de la instancia (*instance*) dentro del array de su situación, para poder localizarlo luego en la función de evaluación; **y *similitud***, es decir, la puntuación de similitud que utilizaremos para comparar dicha instancia con el resto de las que ya están en *similars*.

Finalmente, la función de similitud en sí se ideó como un **vector de pesos** que sumara las diferentes variables utilizadas para comparar instancias usando un peso específico para cada una. En cuanto a los pesos, a pesar de que tratamos de dar mayor peso a aquellos atributos que nos parecían más relevantes a la hora de distinguir las situaciones, como *isEnemy* o *isBlocked*, al final resultó que **Mario evitaba más obstáculos y enemigos poniendo el mismo peso a todos los atributos**. Por tanto, la función de similitud es una función con la suma de todos los atributos, teniendo estos los siguientes valores (sabiendo que todos son nominales):

- Si el atributo a evaluar de ambas instancias tienen el mismo valor, valdrá 1 en la función de similitud.
- Si son distintos, será 0.

Por tanto, la **función** quedaría tal que:

$$isOnGround + jumped + isEnemy + isBlocked + isCoinMushroom$$

7.3. Función de evaluación

El **objetivo** de la función de evaluación es, una vez seleccionadas las instancias más similares a la que se está analizando actualmente en tiempo real, **calcular qué evaluación tiene cada una de las opciones e indicarle a Mario que ejecute la acción de aquella que haya obtenido un mejor resultado**.

Esta función **se aplica en exclusiva a las instancias pertenecientes a la base de conocimiento**, ya que sólo se puede evaluar una instancia si se conoce información del futuro de ella, es decir, qué efectos ha tenido la acción elegida en esa situación concreta. Por esta misma razón, caemos en una situación similar a la que enfrentamos en la función de pertenencia: se puede implementar de dos maneras diferentes.

- La primera opción, que es la que hemos elegido, tiene su base en que los atributos necesarios para la evaluación ya se conocen durante el período de recogida de instancias. Debido a esto, **el resultado de la función de evaluación se puede obtener antes de la toma de decisiones en tiempo real**; justo antes de escribirla en la base de conocimiento, para no calcular evaluaciones de instancias que no se van a utilizar después. Esto tiene un **impacto positivo en el tiempo que disponemos para comparar con las instancias de la base de conocimiento en la función de similitud** (lo que supondría poder incluir más ejemplos para cada situación), ya que nos ahorramos el cálculo de la función de evaluación de todas las instancias que haya elegido como más similares. De este modo, la evaluación de cada una de las instancias de la base de conocimiento se pasa por atributo.
- Por otro lado, la segunda opción sería **calcular el resultado de la función de evaluación durante la toma de decisiones en tiempo real**. Esta opción la desechamos desde el principio, ya que es completamente ineficiente y limita el aprendizaje de Mario (tendríamos que incluir menos ejemplos en la base de conocimiento).

En cuanto a **la función de evaluación** propiamente dicha, esta **ha sido ideada como un vector de pesos**. Dicho vector de pesos suma cada uno de los atributos de evaluación (ponderado si lo requiere) multiplicado por su propio peso. El número de pesos a repartir entre todos los atributos es de 100.

Ya que no se indicó en el enunciado cuál de los dos objetivos tenía más importancia, si avanzar o maximizar la *intermediate reward*, **hemos considerado ambos igual de relevantes en nuestra función de evaluación**. Por ello, a cada uno de los objetivos se les ha asignado 50 puntos del vector de pesos.

Evaluación de la distancia recorrida

Para este objetivo sólo hemos almacenado un atributo en cada una de las instancias: ***distancePassedPhys***.

A diferencia de los atributos que vamos a utilizar para el objetivo de la *intermediate reward*, que suelen encontrarse entre el 0 y el 5; a excepción de *marioStatus*, cuyos valores posibles son 0, -1 y 1 (no hemos visto ningún caso en el que Mario gane o pierda 2 vidas dentro del rango de 6 ticks); **sus valores son números enteros entre 0 y 33**, siendo este último la máxima distancia que puede recorrer Mario durante el período de evaluación.

Si multiplicáramos el valor de *distancePassedPhys* por 50 directamente, Mario siempre se decantaría por aquellas instancias que le hacen avanzar lo máximo posible, obviando la *intermediate reward* siempre. Esta la razón por la que decidimos **ponderar este atributo entre 0 y 1**.

La ponderación funciona de la siguiente manera: sabiendo que la distancia máxima que Mario puede recorrer en 6 ticks es 33, **sólo tenemos que dividir el atributo entre 33 para obtener el valor deseado**. En el caso de que *distancePassedPhys* fuera negativo, esto penalizaría a la evaluación de esa instancia, restando puntuación en lugar de sumarla.

Evaluación de la *intermediate reward*

A diferencia del objetivo anterior, tenemos cuatro atributos de que nos sirven para evaluar: ***marioStatus*, *coinsGained*, *mushroomsDevoured* y *killsByStomp***.

Como no sabíamos cómo repartir los pesos aquí, jugamos a *MarioAI* para ver cómo variaba la *intermediate reward* en cada uno de los eventos que consideran estos atributos:

- Si Mario **recibe daño** de un enemigo y por tanto, pierde vida, la *intermediate reward* **se reduce en 42 puntos**.
- Si Mario **gana una moneda**, la *intermediate reward* **aumenta en 16 puntos**.
- Si Mario **recoge un champiñón**, la *intermediate reward* **aumenta en 58 puntos**.
- Si Mario **elimina a un enemigo**, la *intermediate reward* **aumenta 10 puntos**.

Nos hemos basado en estos **datos reales de la *intermediate reward*** para calcular los pesos de cada uno de los atributos que hemos considerado para evaluar, ya que es el valor total de la misma lo que debemos maximizar. Por tanto, teniendo en cuenta que tenemos 50 puntos a repartir entre estos atributos, proporcionalmente:

- **Perder una vida tiene un peso de 17 puntos**.
- **Ganar una moneda tiene un peso de 6 puntos**.
- **Recoger un champiñón tiene un peso de 23 puntos**.
- **Eliminar a un enemigo tiene un peso de 4 puntos**.

Finalmente, tenemos que considerar si multiplicar los valores de los atributos por sus pesos correspondientes o aplicarles alguna ponderación o filtro, como con *distancePassedPhys*. Respecto a esto, el único problema que hemos observado es que **perder una vida es proporcionalmente menos relevante que obtener un champiñón** (que puede derivar en recuperar vida). Sin embargo, nos parece que perder una vida es lo más grave que le puede pasar a Mario (es muchísimo más fácil perder vida que recuperarla) y es por ello que lo queremos evitar a toda costa. La **solución** más adecuada a esta cuestión pasa, bajo nuestro punto de vista, por **multiplicar su peso por 2**. Así sería el factor más relevante a tener en cuenta en este objetivo.

Otro punto importante a tener en cuenta es que ***marioStatus* puede ser un número positivo si el agente no tiene la vida máxima y encuentra un champiñón**. Esto crearía un conflicto con el atributo *mushroomsDevoured*. Para solucionarlo, pensamos en **incluir a *marioStatus*** en la función de evaluación **sólo en aquellas situaciones en las que fuera menor que cero**. De este modo no premiamos el hecho de recoger un champiñón por dos, ya que el juego real no lo hace y así *marioStatus* sólo sirve para restar a la función de evaluación.

Resumiendo, la **función de evaluación** sería la siguiente:

If *marioStatus* <0:

$$50 \cdot \frac{distancePassedPhys}{33} + 2 \cdot 17 \cdot marioStatus + 6 \cdot coinsGained + 23 \cdot mushroomsDevoured + 4 \cdot killsByStomp$$

Else:

$$50 \cdot \frac{distancePassedPhys}{33} + 6 \cdot coinsGained + 23 \cdot mushroomsDevoured + 4 \cdot killsByStomp$$

Podemos encontrar la implementación de la función de evaluación que hacemos durante la recogida de instancias dentro de la función ***evaluate()***.

La función de evaluación durante la toma de decisiones en tiempo real

El hecho de quitarle trabajo a la función de evaluación durante la ejecución propiamente dicha de la fase de *IBL*, no significa que no tenga nada que hacer durante esta etapa. Sin embargo, su implementación es mucho más sencilla.

Lo que podemos encontrar en ***func_evaluation***, que es el método que lo implementa, es un método que recibe tanto la situación a la que pertenece la instancia, pertenencia, como un array compuesto de diez *instance_similitud*, es decir, las diez instancias que la función de similitud ha encontrado más parecidas a la que estamos evaluando. Por tanto, la función aquí de la evaluación es clara: **recorrer el array de las instancias más similares para comprobar aquella que posee una mejor evaluación** y que, en teoría, sería la más óptima. Para ello, es necesario usar el atributo *id* del array de instancias similares, *similars*; ya que este nos indica la posición absoluta dentro del array de la situación que nos indique *pertenencia*. No podríamos acceder de otra forma a su valor de evaluación, guardado en *eval*. Finalmente, tras elegir la instancia más adecuada, *func_eval* sólo tiene que **devolver su acción correspondiente**.

8. Evaluación de los resultados

Para evaluar el comportamiento de bot generado en esta práctica se llevó a cabo un **experimento de 25 niveles** que no incluimos para la recogida de instancias. Estos comprenden **desde la semilla 1001 a la 1026**.

La primera gráfica se corresponde con una **evaluación sobre la distancia recorrida** por el agente. Debido a que ***BaselineAgent* y *P2BotAgent*** tienen un factor aleatorio, para realizar la medición de la distancia recorrida en un nivel se utilizó la **media de 5 intentos** sobre esta semilla.

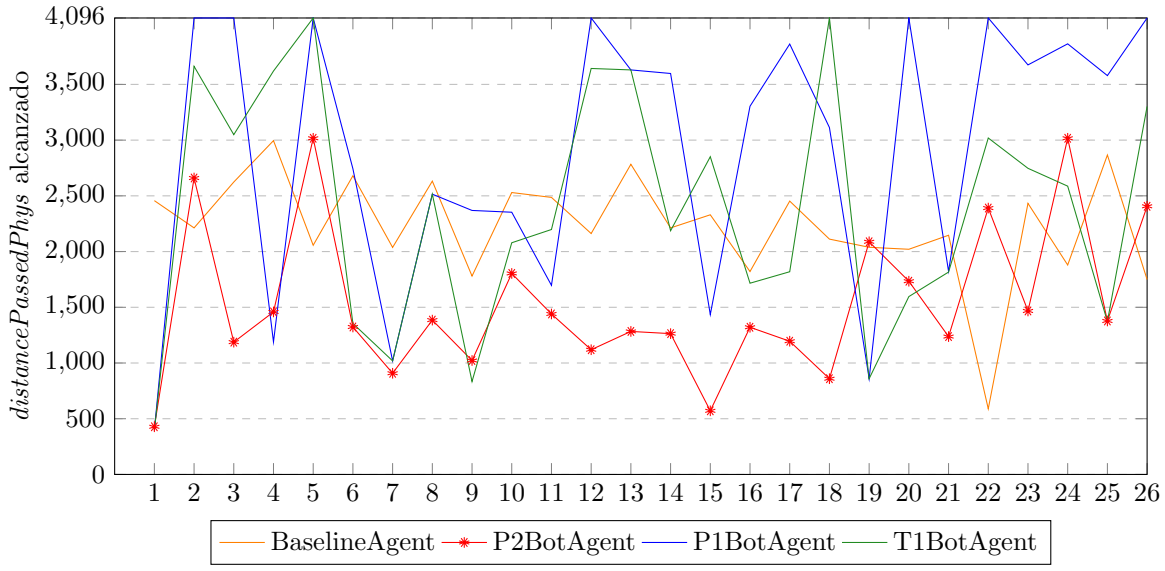

Figura 10: Comparación de agentes en semillas nunca antes entrenadas (**distancia recorrida**)

En la figura se puede observar que *P2BotAgent* (rojo) solo resulta mejor que *BaselineAgent* (naranja) en un **24** % de las veces. **No es un resultado muy positivo** si además consideramos que, al igual que *BaselineAgent*, no consigue superar ningún nivel.

La segunda gráfica se corresponde con una **evaluación sobre la recompensa recogida** por el agente. Al igual que en la evaluación anterior, se realizaron 5 intentos en *BaselineAgent* y *P2BotAgent* para cada una de las semillas y el resultado que se encuentra en la gráfica es una media de ellas. La **recompensa** medida es siempre la **del tick anterior a finalizar la partida**; antes de que se penalice la muerte de Mario o se recompense enormemente su victoria, ya que sólo queremos saber la *intermediate reward* que ha ido consiguiendo durante la partida.

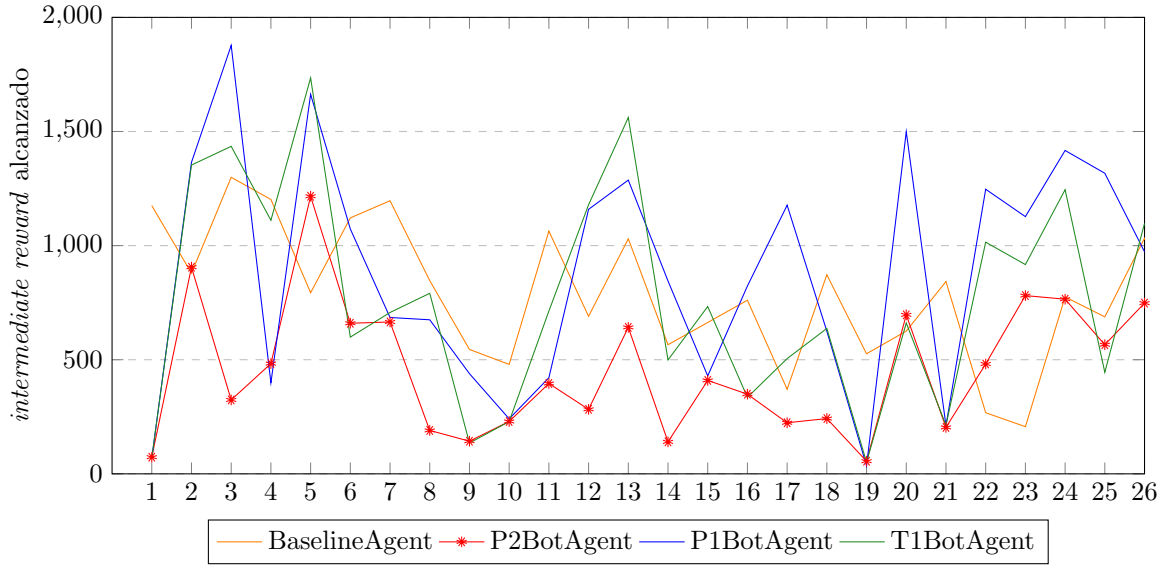

Figura 11: Comparación de agentes en semillas nunca antes entrenadas (**recompensa**)

En la figura se puede observar que *P2BotAgent* (rojo) sólo resulta mejor que *BaselineAgent* (naranja) en un **24** % de los casos. Con respecto al bot del Tutorial 1 (verde) es mejor en un **36** % de los intentos.

Prestando una mayor atención a los niveles 1, 6 y 25 de la **primera gráfica**, podemos ver como el agente empata con el bot del Tutorial 1 y/o de la Práctica 1. Esto se debe a que **Mario se atasca en estos niveles** y hay que hacer movimientos muy específicos para poder avanzar. Ninguno de nuestros tres bot hasta la fecha han sido capaces de superarlos.

Estos resultados no son muy positivos, pero tras analizar el comportamiento de Mario, vimos que su **principal error se basa en bloquearse frente a un obstáculo cuando más adelante existe un enemigo moviéndose**. Si este no es el caso, avanza con normalidad aunque puede mostrar algunas dificultades mucho más leves, como tardar un rato en superar los obstáculos. Lo que le lleva a no superar el nivel en la mayoría de los intentos son tres choques contra enemigos que, a veces, el bot no salta a tiempo.

A raíz de los aspectos comentados en el párrafo anterior se explican los resultados obtenidos en la evaluación de la recompensa obtenida. La recompensa, si se evita recibir daño, está directamente relacionada con la distancia recorrida, por lo que, **a mayor distancia recorrida, mayor es la probabilidad de incrementar la recompensa intermedia pues más monedas se pueden recoger**. También explica la falta de recompensa obtenida el hecho, de que, a pesar de tomar datos tanto para Mario pequeño como grande, ninguno de los dos logró a aprender a saltar para coger una moneda por encima de él. Esto se podría haber solucionado tratando los datos para la situación *Mario tiene una moneda o champiñón que da vida cerca*, pero con el que hicimos nos pareció suficiente.

Como comentamos anteriormente, estamos bastante seguros que **el mayor problema de este bot está relacionado con la manera de recoger instancias debido al corto tiempo de reacción con el que cuenta el ser humano a cargo de recogerlas**. Cuando el agente que las recoge es el bot, los resultados son mucho más coherentes, pero no se pueden utilizar por no intentar maximizar la *intermediate reward*.

9. Conclusiones

En cuanto al desarrollo de la práctica, podemos destacar que la fase de implementar las funciones especificadas no era muy complicada, lo que nos parece una gran ventaja de este modelo. Sin embargo, llegar hasta sus versiones finales no fue sencillo, sobre todo con la función de similitud.

En cuanto a la fase de aprendizaje y recogida de instancias, hemos podido ver la importancia de las situaciones recogidas y la manera de obtenerlas. Pensar las situaciones a recoger de un humano resultó ser una tarea más difícil de lo que en principio estimamos. Esto se debe a dos motivos: la corta capacidad de reacción de un humano frente a un bot, y la capacidad de realizar muchas más acciones que las que podría ejecutar el bot de la Práctica 1.

Sin embargo, también pudimos ver que las instancias recogidas servían para guiar a Mario a través del mapa, aunque no desde el principio: tras un rápido proceso de limpieza de la base de conocimiento, Mario pasó de no saltar nunca a evitar enemigos y a superar obstáculos.

Aunque el agente resultante no obtiene los resultados esperados, gracias a esta práctica hemos aprendido importantes nociones relacionadas con este tipo de aprendizaje. Es muy ilustrativo poder ver cómo funciona lo que vemos en las clases teóricas, ya que estas tienen a menudo conceptos difíciles de entender.

10. Comentarios personales

A lo largo de esta práctica hemos podido ver de primera mano cómo funciona un proceso de aprendizaje basado en instancias.

Somos conscientes de que aún existe espacio de mejora para el bot generado y nos habría gustado seguir explorando estas opciones, pero el tiempo ha resultado ser un factor limitante debido a que estas semanas son críticas para varias asignaturas de las que cursamos. De todos modos, estamos satisfechos con nuestro trabajo ya que nos ha llevado a comprender en profundidad la teoría detrás de este modelo de aprendizaje. Asímismo, también nos ha dotado de la experiencia de lidiar con este tipo de problemas.

En resumen, creemos que esta práctica es muy adecuada con respecto a lo que se enseña en la asignatura. Además, nos ha ayudado a ganar seguridad y confianza a la hora de afrontar tareas de aprendizaje similares en un futuro, ya que hemos sido testigos de los factores más determinantes de *IBL* y seremos capaces de explotarlos adecuadamente la próxima vez.