Práctica 2:
Aprendizaje basado en instancias

Grado en Ingeniería Informática

Aprendizaje Automático

Aitor Alonso Núñez NIA 100346169 Gr. 83
100346169@alumnos.uc3m.es
Daniel Gak Anagrov NIA 100318133 Gr. 83
100318133@alumnos.uc3m.es

Aitor Alonso Núñez NIA 100346169
Daniel Gak Anagrov NIA 100318133

uc3m | Universidad Carlos III de Madrid

Índice

Aitor Alonso Núñez NIA 100346169
Daniel Gak Anagrov NIA 100318133

uc3m | Universidad Carlos III de Madrid

Introducción

Este documento sirve como memoria de la práctica 2: *Aprendizaje basado en instancias*, de la asignatura Aprendizaje Automático. En la misma, hemos tenido que programar un agente basado en instancias, esto es, que aprenda a jugar al Mario mediante comparación y evaluación del momento del juego actual contra momentos almacenados en una base de conocimiento.

A lo largo del presente documento se describirá el *Tratamiento de datos*, donde comentamos todo lo referente a la toma de ejemplos y carga inicial de la base de conocimiento; *Funciones*, donde se describen y detallan las funciones utilizadas para clasificar y evaluar instancias; y por último, una *Evaluación*, *Conclusiones* y *Comentarios personales*

Tratamiento de datos

Toma de ejemplos

Para la toma de ejemplos hemos utilizado únicamente el agente humano `P2HumanAgent`, ya que consideramos que el agente a programar aprende por imitación, mediante comparación y evaluación de las instancias, por lo que una toma de ejemplos de un comportamiento humano sería bastante adecuada por los siguientes motivos.

El comportamiento humano, en este contexto, realiza acciones con sentido. Esto es así puesto que somos personas que sabemos jugar al Mario, conocemos los objetivos a puntuar en esta práctica (acabar el nivel con la mayor recompensa), y realizamos las acciones necesarias en cada instante del juego para lograr tal fin.

Si bien el comportamiento humano tiene cierto grado de aleatoriedad, puesto que no efectúa la misma acción de la misma manera en dos situaciones idénticas (ya sea por tiempo de reacción del agente humano, estrategias a largo plazo, etc.), en general la acción realizada tiene sentido, es correcta, y va enfocada a conseguir el objetivo final anteriormente descrito.

Luego, puesto que nuestro agente inteligente a programar aísla cada instancia de cada momento del juego, y la analiza en base a valores objetivos (sin contexto anterior o futuro), consideramos que elimina la aleatoriedad humana del aprendizaje, quedándose solo con lo importante (situación $>$ acción $>$ evaluación).

Estrategia de selección de instancias

Puesto que posteriormente en nuestro agente vamos a evaluar las instancias de nuestra base de conocimiento en base a la recompensa futura que se consiguió realizando la acción relacionada con cada instancia, conviene evitar instancias de entrenamiento (que acabarán posteriormente en la base de conocimiento) que contengan acciones que proporcionaron recompensa muy a futuro, ya que podría meter ruido y dificultar la evaluación.

Un ejemplo de este tipo de acciones es la de disparar con Mario. Desde que Mario dispara una bola de fuego, pueden pasar varios segundo de juego hasta que un enemigo es eliminado, y por tanto hasta que se consigue la recompensa por ello. Puesto que un segundo de juego son 24 ticks, y vamos a evaluar las instancias en base a la recompensa que conseguimos en torno a 6 – 24 ticks después de realizar la acción correspondiente, es mejor ignorar estas situaciones.

Para directamente evitar este tipo de instancias a la hora de la toma de ejemplos, hemos desactivado en nuestro agente humano `P2HumanAgent` la posibilidad de pulsar la tecla "S", correspondiente con disparar y correr. Así, aunque el jugador presione la tecla "S", el agente no está capturando el evento, pues hemos eliminado el código correspondiente a dicha tecla de la función `getAction()`.

Atributos

Nuestras instancias de entrenamiento, que han sido tomadas con el agente humano `P2HumanAgent` con ayuda del código de toma de datos contenido en el fichero `tools/TrainingFile.java`, consta de los siguientes atributos en este orden, si bien no todos son utilizados posteriormente en el agente bot inteligente, producto de esta práctica.

Atributos para ayudar a localizar o identificar la instancia de entrenamiento

1. **timeSpent:** Tiempo gastado en segundos del juego (`short`)
2. **timeLeft:** Tiempo restante en segundos del juego (`short`)

Atributos relacionados con la intermediateReward

3. **intermediateReward:** Recompensa actual en el momento de la instancia (`short`)
4. **intermediateRewardWonLastTick:** Recompensa obtenida respecto al tick anterior (1/24s) (`short`)
5. **intermediateRewardWonLast6Ticks:** Recompensa obtenida respecto a los 6 ticks anteriores (1/4s) (`short`)
6. **intermediaRewardWonFuture6Ticks:** Recompensa obtenida 6 ticks después (1/4s) de ejecutar la acción de esta instancia (`short`)
7. **intermediaRewardWonFuture12Ticks:** Recompensa obtenida 12 ticks después (1/2s) de ejecutar la acción de esta instancia (`short`)
8. **intermediaRewardWonFuture24Ticks:** Recompensa obtenida 24 ticks después (1s) de ejecutar la acción de esta instancia (`short`)

Atributos relacionados con la matriz de observación (mergeObs)

9. **nearestEnemyLeftDistance:** Distancia euclídea al enemigo más cercano a la izquierda de Mario (`float`)
10. **nearestEnemyLeft_X:** posición X de la mergeObs del enemigo más cercano por la izquierda (`byte`)
11. **nearestEnemyLeft_Y:** posición Y de la mergeObs del enemigo más cercano por la izquierda (`byte`)
12. **nearestEnemyRightDistance:** Distancia euclídea al enemigo más cercano a la derecha de Mario (`float`)
13. **nearestEnemyRight_X:** posición X de la mergeObs del enemigo más cercano por la derecha (`byte`)
14. **nearestEnemyRight_Y:** posición Y de la mergeObs del enemigo más cercano por la derecha (`byte`)
15. **nearestBlockLeftDistance:** Distancia euclídea al bloque más cercano a la izquierda de Mario (`float`)

16. **nearestBlockLeft_X:** posición X de la mergeObs del bloque más cercano por la izquierda (`byte`)
17. **nearestBlockLeft_Y:** posición Y de la mergeObs del bloque más cercano por la izquierda (`byte`)
18. **nearestBlockRightDistance:** Distancia euclídea al bloque más cercano a la derecha de Mario (`float`)
19. **nearestBlockRight_X:** posición X de la mergeObs del bloque más cercano por la derecha (`byte`)
20. **nearestBlockRight_Y:** posición Y de la mergeObs del bloque más cercano por la derecha (`byte`)
21. **nearestCoinLeftDistance:** Distancia euclídea a la moneda más cercana a la izquierda de Mario (`float`)
22. **nearestCoinLeft_X:** posición X de la mergeObs de la moneda más cercana por la izquierda (`byte`)
23. **nearestCoinLeft_Y:** posición Y de la mergeObs de la moneda más cercana por la izquierda (`byte`)
24. **nearestCoinRightDistance:** Distancia euclídea a la moneda más cercana a la derecha de Mario (`float`)
25. **nearestCoinRight_X:** posición X de la mergeObs de la moneda más cercana por la derecha (`byte`)
26. **nearestCoinRight_Y:** posición Y de la mergeObs de la moneda más cercana por la derecha (`byte`)
27. **numEnemiesObserved:** número total de enemigos identificados en la matriz de observación (`short`)
28. **numCoinsObserved:** número total de monedas identificadas en la matriz de observación (`short`)

Atributos booleanos o binarios

29. **enemyNearRight:** valor numérico binario (0-1) que indica si hay (1) o no (0) enemigos en una submatriz de la matriz de observación ([fil][col] == [8-9][10-11]) (`byte`)
30. **blockNearRight:** valor numérico binario (0-1) que indica si hay (1) o no (0) obstáculos en una submatriz de la matriz de observación ([fil][col] == [8-9][10-11]) (`byte`)
31. **enemyAheadOnFloorHeight:** valor numérico binario (0-1) que indica si hay (1) o no (0) enemigos en una submatriz de la matriz de observación ([fil][col] == [10][10-11]) (`byte`)
32. **blockAheadOnFloorHeight:** valor numérico binario (0-1) que indica si hay (1) o no (0) obstáculos en una submatriz de la matriz de observación ([fil][col] == [10][10-11]) (`byte`)
33. **abyssAhead:** valor numérico binario (0-1) que indica si hay (1) o no (0) foso/abismo/acantilado delante (en columna [10]) (`byte`)
34. **isMarioOnGround:** valor numérico binario (0-1) que indica si Mario está (1) o no está (0) en el suelo (`byte`)
35. **isMarioAbleToJump:** valor numérico binario (0-1) que indica si Mario puede (1) o no puede (0) saltar (bloqueo de salto) (`byte`)
36. **isMarioAbleToShoot:** valor numérico binario (0-1) que indica si Mario puede (1) o no puede (0) disparar (`byte`)

37. **isMarioCarrying:** valor numérico binario (0-1) que indica si Mario está (1) o no está (0) llevando un caparazón (`byte`)
38. **enemyWasKilledBin:** valor numérico binario (0-1) que indica si Mario si (1) o no (0) ha muerto un enemigo en la instancia actual (`byte`)
39. **marioWasInjuredBin:** valor numérico binario (0-1) que indica si Mario si (1) o no (0) ha ha sido herido Mario en la instancia actual (`byte`)
40. **isSlopeDown:** valor numérico binario (0-1) que indica si hay (1) o no (0) un foso/abismo/acantilado delante (en matriz de observación [16][10] no hay bloque) (`byte`)

Otros atributos

41. **coinsGainedLastTick:** número de monedas conseguidas respecto al tick anterior (`byte`)
42. **marioMode:** modo actual de Mario: 0,1,2 == pequeño, grande, fuego (`byte`)
43. **marioStatus:** estado actual de Mario: 0,1,2 == muerto/derrota, victoria, corriendo/jugando (`byte`)
44. **actionKey:** acción o acciones realizadas durante esta instancia, codificada como un valor numérico que diferencia varias situaciones (`byte`)
    - Ninguna acción o tecla pulsada: **0**
    - Salto (tecla A): **1**
    - Derecha (tecla flecha derecha): **2**
    - Derecha-salto (teclas flecha derecha + A): **3**
    - Derecha-disparar (teclas flecha derecha + S): **4** (no puede existir en esta toma de ejemplos)
    - Derecha-salto-disparar (teclas flecha derecha + A + S): **5** (no puede existir en esta toma de ejemplos)
    - Izquierda (tecla flecha izquierda): **6**
    - Izquierda-salto (teclas flecha izquierda + A): **7**
    - Izquierda-disparar (teclas flecha izquierda + S): **8** (no puede existir en esta toma de ejemplos)
    - Izquierda-salto-disparar (teclas flecha izquierda + A + S): **9** (no puede existir en esta toma de ejemplos)

Como se puede observar, el tipo de cada atributo se ha elegido buscando ocupar la menor memoria posible, dado que el tipo de agente a programar hace un uso intensivo de memoria principal.

Preprocesado

Desde la toma de ejemplos hasta la carga en memoria del agente por instancias no se han modificado ni aplicado ningún preprocesado a los datos, si bien hay que recordar que hemos evitado capturar las instancias en las que Mario dispara imposibilitando que el agente humano realice esta acción.

Carga de datos en memoria

Para la carga de las instancias de entrenamiento en memoria del agente, con el fin de conformar la base de conocimiento del aprendizaje basado en instancias, nos hemos ayudado de una clase objeto auxiliar a la que hemos llamado `tools/Instance.java`. Esta clase, que representa una instancia, posee todos los atributos y tipos especificados en el apartado anterior *Atributos*.

La carga se ha realizado en el constructor del agente. La misma consiste en leer las instancias de entrenamiento de un fichero `knowledge_base.csv` (que es el `.arff` generado con `P2HumanAgent` al que se le ha eliminado la cabecera del weka), llamar a la función de pertenencia, y almacenar la instancia en el `ArrayList` correspondiente con la situación identificada en el apartado siguiente *Situaciones identificadas*. Estos `ArrayList` son atributos del agente y por tanto variables globales.

Situaciones identificadas

Para determinar las situaciones se ha realizado una experimentación en diferentes niveles, de forma que hemos observado que las situaciones que obtienen mejores resultados[1] son las cinco siguientes:

- **coinsNearClass:** Pertenecen únicamente a esta clase todas aquellas instancias en las que en la matriz de observación se observan monedas sin observarse enemigos.
- **enemiesNearClass:** Pertenecen únicamente a esta clase todas aquellas instancias en las que en la matriz de observación se observen enemigos sin observarse monedas.
- **coinsEnemiesClass:** Pertenecen únicamente a esta clase todas aquellas instancias las que en la matriz de observación se observen enemigos y monedas a la vez.
- **marioOnAirClass:** Pertenecen únicamente a esta clase todas aquellas instancias en las que mario no se encuentra en el suelo. Esta clase predominará sobre las anteriores, una situación en la que mario está en el aire y hay enemigos o monedas será considerada como perteneciente a la clase `marioOnAirClass`
- **defaultClass:** Pertenecen únicamente a esta clase aquellas instancias que no han podido ser clasificadas conforme a los criterios anteriores (ej: Mario está en el suelo y no hay enemigos ni monedas en la matriz de observación).

[1] Para observar resultados de una configuración determinada de MarioAI, henos utilizado el script `estadísticas.sh` utilizado en la práctica P1, que procesa varios niveles de mario sin interfaz dando como resultado datos como el número y porcentaje de niveles superado, media de longitud completada por nivel, etc. Típicamente se han utilizado 100 niveles para ver resultados y comparar configuraciones. Todo ello se describe con mayor profundidad en el apartado de **??**.

Funciones

Como estamos introduciendo conocimiento del dominio a la hora de crear las funciones, especialmente en la función de pertenencia, no es totalmente necesario –como dijo Jose Carlos en clase– hacer una función euclídea con pesos, por lo que optaremos por esta opción y la función de pertenencia quedarán implementada con uso de operadores `if-else`.

No obstante, cualquier estructura `if-else` puede quedar implementada matemáticamente mediante el uso de mínimos y máximos, de forma que la siguiente condición:

$$if(\ i.numCoinsObserved > 0\ ) \quad return\ 2;$$

Siendo el atributo `numCoinsObserved` siempre ≥ 0 (como es el caso de todos nuestros atributos menos las fluctuaciones de `intermediateReward`), puede traducirse en:

$$return \quad min(\ 1\ ,\ i.numCoinsObserved\ ) * 2;$$

Así pues, optamos por la implementación `if-else` –tras comentarlo Jose Carlos en clase– por un motivo de claridad y mantenimiento del código. Asimismo, algunos valores del resto de funciones (que sí son euclídeas con pesos) quedarán complementados haciendo uso de otros operadores condicionales (ej: `(<condición>)?<valorTrue>:<valorFalse>`).

Función de pertenencia

Como se ha comentado anteriormente, la función se ha implementado en forma de `if-else`, de manera que en el constructor del agente se distribuyen todas las instancias creadas en la fase de *Tratamiento de datos* insertándolas en un `ArrayList` (como indicamos en *Carga de datos en memoria*) por cada situación identificada en el apartado de *Situaciones identificadas*. La estructura de los `if-else` tiene la siguiente forma, en este orden:

Para cada instancia:

1. Si el atributo `isMarioOnGround` es 0 entonces la instancia se almacena en `marioOnAirClass`.
2. Si el atributo `numCoinsObserved` es mayor que 0 y `numEnemiesObserved` es menor o igual que 0, entonces la instancia se almacena en `coinsNearClass`.
3. Si el atributo `numCoinsObserved` es menor o igual que 0 y `numEnemiesObserved` es mayor que 0, entonces la instancia se almacena en `enemiesNearClass`.
4. Si tanto el atributo `numCoinsObserved` como `numEnemiesObserved` son ambos mayores que 0, entonces la instancia se almacena `coinsEnemiesClass`.
5. Por defecto, si no se ha obtenido un resultado verdadero en ninguna de las condiciones anteriores, la instancia se almacena en `defaultClass`

Los atributos anteriormente nombrados han quedado descritos con anterioridad en el apartado *Atributos*.

Se ha probado esta función sobre diferentes cantidades de instancias y la distribución de las instancias en las clases es la siguiente:

Clase	% del total de instancias
coinsNearClass	~2 %
enemiesNearClass	~10 %
coinsEnemiesClass	~30 %
marioOnAirClass	~55 %
defaultClass	~1 %

Figura 1: Distribución de las instancias en las clases mediante el uso de la función de pertenencia.

Las descripciones de las clases se pueden encontrar en el apartado *Situaciones identificadas*.

Función de similitud

La función de similitud que finalmente hemos implementado, una función matemática euclídea con pesos, en primer lugar identifica la pertenencia de la situación del tick actual del juego a alguna de las situaciones descritas anteriormente en *Situaciones identificadas* con los mismos criterios que se utilizan en *Función de pertenencia*.

A continuación, itera únicamente sobre las instancias del `ArrayList` correspondiente a dicha clase/situación y calcula la distancia de cada una de ellas a la instancia del tick actual del juego. Para ello, se suman las diferencias en valor absoluto de todos los atributos de los que consta la instancia, con excepción de los atributos de `time*`, `infiniteReward*` y `actionKey`. Esto es posible puesto que todos nuestros atributos son de tipo numérico.

Finalmente, la función de similitud retorna un array de 20 posiciones con las 20 instancias más similares, es decir, aquellas con menor distancia (menor valor de la función de similitud, que son sumas) a la instancia del tick actual del juego.

Queremos remarcar que hemos tomado una decisión de diseño para el caso en que dos instancias de la base de conocimiento tienen la misma distancia respecto a la instancia del tick actual del juego, en la que priorizamos aquella que contiene la acción salto o derecha-salto con el objetivo de evitar situaciones en las que Mario se queda atascado yendo hacia la derecha contra un muro.

Función de evaluación

La función de evaluación que finalmente hemos implementado, una función matemática euclídea con pesos, es la siguiente:

$$evaluation = 0.01 \cdot intermediaRewardWonFuture6Ticks + \ 0.02 \cdot intermediaRewardWonFuture12Ticks$$

$$+ 0.04 \cdot intermediaRewardWonFuture24Ticks + \ actionWeight;$$

Donde `actionWeight` es un peso asignado en función de la tecla pulsada que se calcula en el código de la función de evaluación. Su valor será:

- Ninguna acción o ninguna tecla pulsada: peso 0.
- Salto (tecla A): peso 100.
- Derecha (tecla flecha derecha): peso 20.
- Derecha-salto (teclas flecha derecha + A): peso 600. [2]
- Derecha-disparar (teclas flecha derecha + S): peso 20. [3]
- Derecha-salto-disparar (teclas flecha derecha + A + S): peso 600.
- Izquierda (tecla flecha izquierda): -10 * `timeSpent`. [4]
- Izquierda-salto (teclas flecha izquierda + A): -10 * `timeSpent`.
- Izquierda-disparar (teclas flecha izquierda + S): -10 * `timeSpent`.
- Izquierda-salto-disparar (teclas flecha izquierda + A + S): -10 * `timeSpent`.

El porqué de los distintos pesos o su diferencia se comenta en las notas a pie de página asociadas a cada uno, aunque el valor exacto de estos se ha calculado mediante prueba y error.

Se ha elegido únicamente estos atributos ya que damos importancia solo al refuerzo que consiguió el agente y a la acción que realizó, buscando conseguir los dos objetivos principales de esta práctica: que el agente consiga la mayor recompensa/refuerzo posible, y que el agente avance lo máximo posible o complete el nivel.

Asimismo, para evitar situaciones en las que Mario se quede atascado y la función de evaluación esté devolviendo siempre la misma acción de la misma instancia, si detectamos que Mario permanece en la misma casilla (coordenada X) durante al menos dos segundos, ejecutaríamos la acción de la segunda instancia con mayor evaluación, como se comenta en el código.

Evaluación

Para la obtención de estadísticas a gran escala, hemos utilizado el mismo script en bash que utilizamos durante la práctica 1, `estadísticas.sh`, al que le hemos realizado algunas pequeñas modificaciones por claridad. Este script se incluye junto con el reto de material entregado en esta práctica.

Hemos probado varios pesos en la *Función de evaluación* hasta quedarnos con la versión actual, que conseguía un rendimiento más cercano al bot de la práctica anterior, aunque en esencia sigue siendo menos eficiente que este. A continuación, se realiza una comparación de los resultados del análisis empírico respecto al bot de la práctica 1 y el BaselineAgent proporcionado, usando una muestra de ejecución de 1000 niveles.

[2] Añadiendo más peso al principal movimiento de avance es la forma en la que beneficiamos las instancias que acercan al objetivo del juego. Estas decisiones las tomamos teniendo en cuenta que se permite introducir conocimiento sobre el dominio como hemos explicado al comienzo del apartado *Funciones*.

[3] Al no tener instancias de entrenamiento de la tecla disparar, debido a como hemos indicado en el último párrafo del apartado *Estrategia de selección de instancias*, este tipo de agente programado no tiene nunca la posibilidad de optar por acciones que incluyan disparar, por lo que realmente el peso que asociemos a estas acciones es indiferente.

[4] Penalizamos el movimiento hacia la izquierda haciendo que sea mayor en función al tiempo gastado, para que las instancias de movimiento derecho, tengan mayor preferencia cuanto menos tiempo quede para completar el nivel.

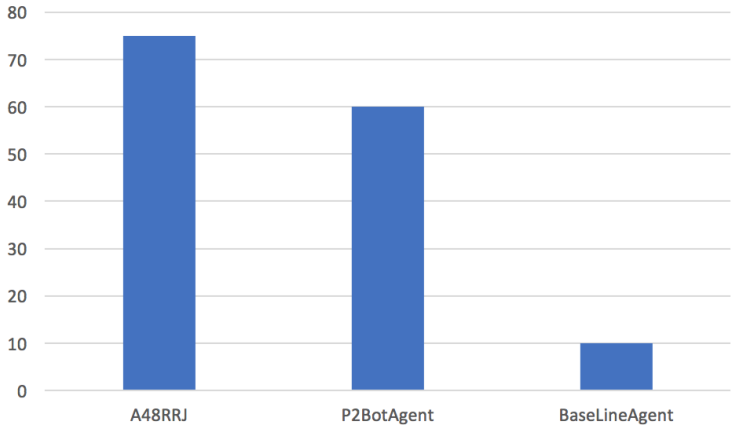

Figura 2: Total de niveles superados sobre 1000

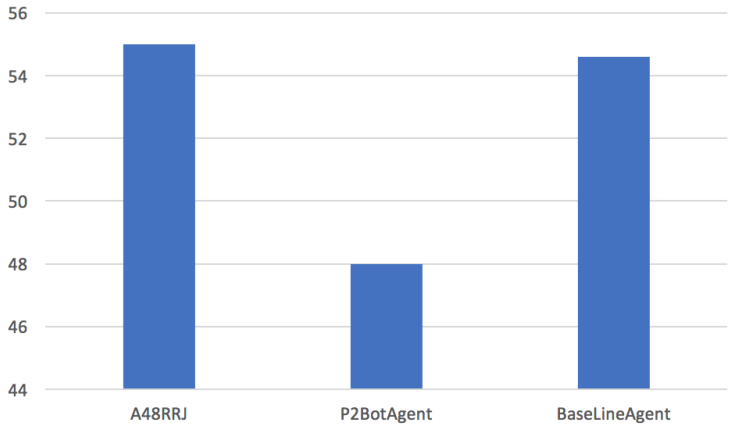

Figura 3: Porcentaje medio de la superación en cada nivel.

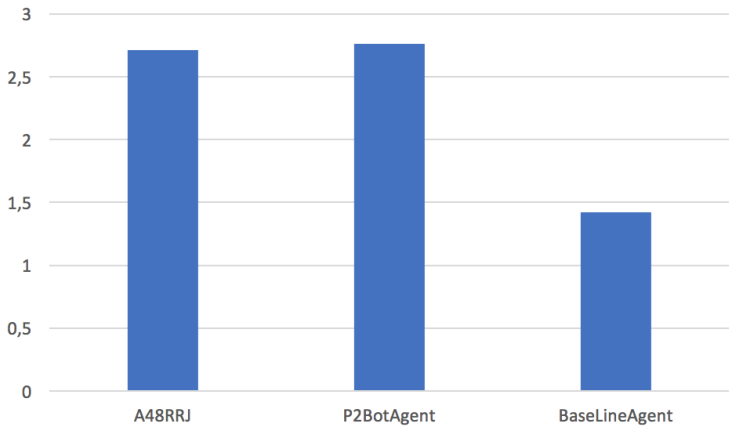

Figura 4: Media de colisiones con enemigo por nivel.

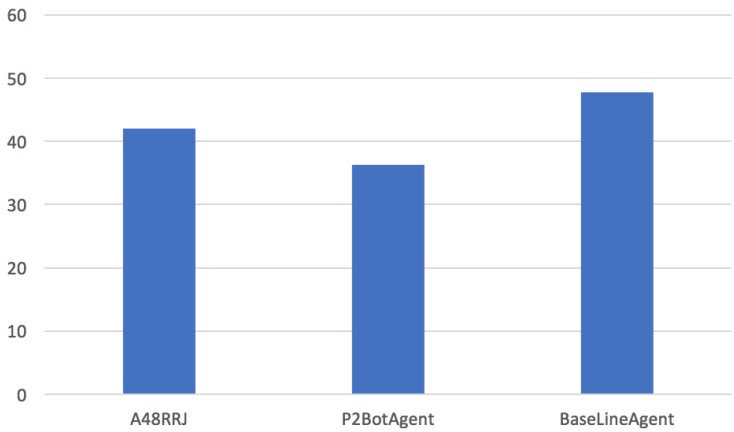

Figura 5: Media de monedas recogidas por nivel.

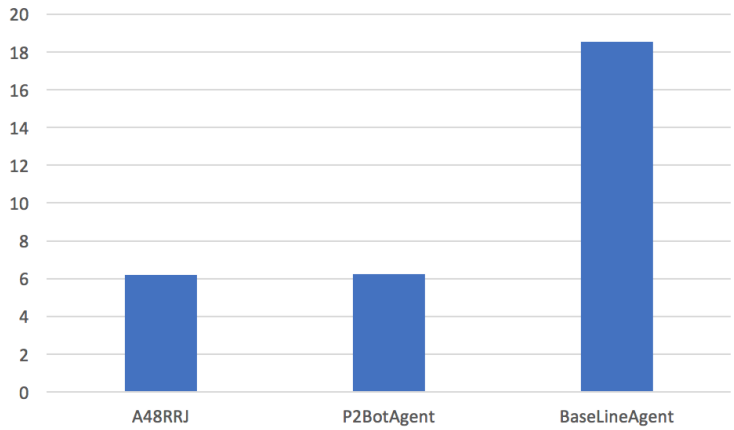

Figura 6: Media de muertes de enemigos por nivel.

Se ha observado durante la experimentación que con una disposición determinada de pesos, nuestro agente conseguido mediante aprendizaje basado en instancias ha conseguido moverse para atrás para esquivar obstáculos complejos, cosa que el bot A48RRJ no sería capaz de hacer. En la versión entregada no se observan estos comportamientos debido a que se el ajuste de pesos se ha realizado para maximizar la eficiencia del bot. A continuación reflejamos dos de estas situaciones complejas en niveles distintos, donde en verde vemos el camino que realiza mario, y en rojo el que debería hacer para salir de esa situación.

Figura 7: Nivel 61

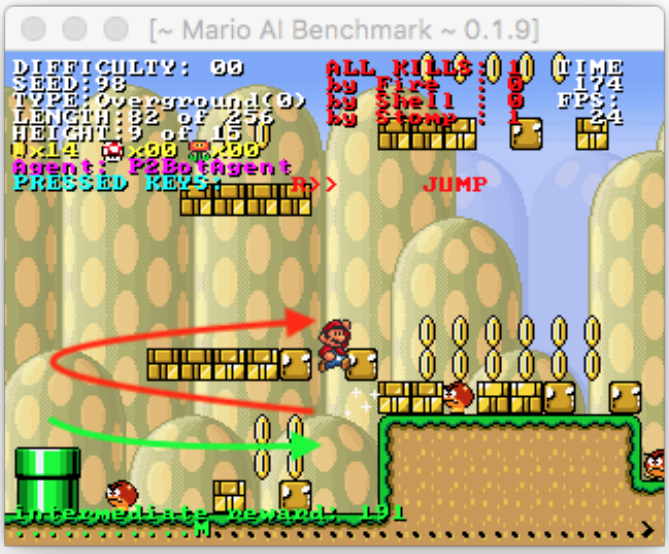

Figura 8: Nivel 98

Descripción del material entregado

Junto a esta memoria, se entrega el siguiente material:

- **estadísticas.sh:** script en bash utilizado para la captura de estadísticas, estudio y evaluación del agente para pequeños y grandes volúmenes de ejecuciones o niveles. Requiere como dependencia python 3. Su ejecución es la siguiente:

```
./estadísticas.sh <agente> <num_niveles>
```

- **weka/ejemplos_entrenamiento/P2HumanAgent_1.arff:** fichero que contiene la toma de ejemplos realizada con el agente humano, en formato arff y con cabecera para su tratamiento con weka.
- **weka/ejemplos_entrenamiento/knowledge_base.csv:** fichero csv que contiene las instancias de entrenamiento que conforman la base de conocimiento del agente. Es el mismo fichero que P2HumanAgent_1.arff pero sin cabecera de weka y con nueva extensión.
- **src/ch/idsia/agents/controllers/human/P2HumanAgent.java:** agente humano con el que se ha realizado la toma de ejemplos de entrenamiento contenida en los dos ficheros anteriormente mencionados.
- **src/ch/idsia/agents/controllers/P2BotAgent.java:** agente bot programado con técnicas de inteligencia artificial de aprendizaje basado en instancias. Principal producto de esta práctica. **Este es el bot programado y no un bot de captura de datos.**
- **src/ch/idsia/tools/TrainingFile.java:** clase java que contiene toda la lógica de captura de ejemplos. Es utilizada por P2HumanAgent.java
- **src/ch/idsia/tools/Instance.java:** clase objeto representación de una instancia, ya sea de entrenamiento o del tick actual del juego. Facilita su tratamiento en el código y su almacenamiento en memoria durante tiempo de ejecución. Es utilizada por P2BotAgent.java

Conclusiones

De la realización de un agente inteligente que utiliza técnicas de aprendizaje automático basado en instancias podemos concluir lo siguiente:

- El agente basado en instancias es mucho menos eficiente en términos de recursos computacionales que el agente basado en árboles o reglas de decisión, ya que la revisión de instancias en cada tick añade una carga computacional adicional muy elevada para obtener una decisión a partir de la entrada.
- Por el mismo motivo que el punto anterior, el bot de esta práctica (aprendizaje basado en instancias) tendrá un comportamiento más deficiente que el bot de la práctica anterior (árboles/reglas de decisión). Esto es debido a que, en este dominio, necesitamos un comportamiento en tiempo real, por lo que debemos entre otras cosas vigilar el número de instancias de nuestra base de conocimiento, intentando que no sea demasiado elevado. Esto puede suponer una mal resultado en la aplicación de la hipótesis del aprendizaje inductivo, como se ha observado en el apartado anterior *Evaluación*

- Este agente trata mucho mejor (aprende mucho mejor) el comportamiento humano, pues al comparar instancias aisladas en base a criterios objetivos, anula con bastante efectividad la parte aleatoria del comportamiento humano, quedándose solo con los aspectos de aprendizaje importantes, como se indicó en el último apartado de *Toma de ejemplos*.

Comentarios personales

Si bien la práctica ha sido entretenida y de una dificultad asequible, nos ha costado bastante obtener las situaciones en las que clasificar las instancias, así como determinar los pesos que aplicar a cada atributos en las funciones matemáticas euclídeas.

Si bien esto es una parte importante de la práctica y algunas decisiones se han tomado en base a criterios objetivos, como se ha detallado en esta memoria o en el código, mucho de estos pesos o clasificaciones se han calculado mediante prueba y error. Por eso, hubiésemos agradecido una guía, una clase, material o algún recurso que nos enseñara cómo calcular estos pesos de forma, si no óptima, al menos más razonada que el típico prueba y error.

Desarrollar estos apartados de la práctica mediante prueba y error nos ha resultado, además de improductivo y contraproducente (ya que no ha aportado conocimiento reseñable per se), una pérdida elevada de tiempo en un momento del cuatrimestre en el que la carga académica es muy elevada para todas las asignaturas.