Práctica 3
Aprendizaje por refuerzo

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

Introducción

Este documento sirve como memoria de la práctica 3: *Aprendizaje por refuerzo*, de la asignatura Aprendizaje Automático. En la misma, hemos tenido que programar un agente que juegue de forma automática a Mario haciendo uso del algoritmo de aprendizaje por refuerzo *Q-learning*, de forma que el dominio ha quedado representado como un espacio de estados discretos, identificando y evaluando diferentes situaciones para obtener una política de actuación a partir de este algoritmo.

A lo largo del presente documento se describirán los siguientes puntos, en el mismo orden que como se ha realizado en esta práctica:

- *Tratamiento de datos*: Este apartado contiene la toma de ejemplos, la descripción de los atributos, la forma de almacenarlos y el preprocesamiento aplicado.
- *Identificación de situaciones*: Explica el critero de creación de situaciones y la forma en la que se identifica si un tick (instancia de un momento de juego) pertenece a ella o no.
- *Q-Learning*: En este apartado queda recogido el cálculo del refuerzo y la aplicación del algoritmo de aprendizaje por refuerzo *Q-Learning*.
- Al final de este documento quedan recogidos la *Evaluación*, la *Descripción del material entregado*, y las *Conclusiones* y *Comentarios personales*

Tratamiento de datos

Atributos

Nuestras instancias de entrenamiento, que han sido tomadas con el agente humano `P3HumanAgent` con ayuda del código de toma de datos contenido en el fichero `tools/TrainingFile.java`, generan una tupla de datos por cada tick de la ejecución del juego que contiene la siguiente información:

- Atributos del tick N.
- Acción ejecutada en el tick N.
- Atributos del tick $N + 12$.
- Refuerzo (cálculo inicial en tiempo de toma de ejemplos finalmente ignorado, ver *Cálculo del rerfuerzo*).

Todas estas tuplas han sido escritas en el fichero `weka/ejemplos_entrenamiento/P3HumanAgent_1.arff` con una cabecera apropiada para su tratamiento con Weka. A continuación se describen los atributos de los que constan estas tuplas.

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 captura 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`). Este atributo no existe para la instancia del tick $N + 12$ (segunda mitad de la tupla).

Atributos relacionados con la matriz de observación (mergeObs)

7. **nearestEnemyLeftDistance:** Distancia euclídea al enemigo más cercano a la izquierda de Mario (`float`)
8. **nearestEnemyLeft_X:** posición X de la mergeObs del enemigo más cercano por la izquierda (`byte`)
9. **nearestEnemyLeft_Y:** posición Y de la mergeObs del enemigo más cercano por la izquierda (`byte`)
10. **nearestEnemyRightDistance:** Distancia euclídea al enemigo más cercano a la derecha de Mario (`float`)
11. **nearestEnemyRight_X:** posición X de la mergeObs del enemigo más cercano por la derecha (`byte`)
12. **nearestEnemyRight_Y:** posición Y de la mergeObs del enemigo más cercano por la derecha (`byte`)
13. **nearestBlockLeftDistance:** Distancia euclídea al bloque más cercano a la izquierda de Mario (`float`)
14. **nearestBlockLeft_X:** posición X de la mergeObs del bloque más cercano por la izquierda (`byte`)
15. **nearestBlockLeft_Y:** posición Y de la mergeObs del bloque más cercano por la izquierda (`byte`)
16. **nearestBlockRightDistance:** Distancia euclídea al bloque más cercano a la derecha de Mario (`float`)
17. **nearestBlockRight_X:** posición X de la mergeObs del bloque más cercano por la derecha (`byte`)
18. **nearestBlockRight_Y:** posición Y de la mergeObs del bloque más cercano por la derecha (`byte`)
19. **nearestCoinLeftDistance:** Distancia euclídea a la moneda más cercana a la izquierda de Mario (`float`)
20. **nearestCoinLeft_X:** posición X de la mergeObs de la moneda más cercana por la izquierda (`byte`)
21. **nearestCoinLeft_Y:** posición Y de la mergeObs de la moneda más cercana por la izquierda (`byte`)
22. **nearestCoinRightDistance:** Distancia euclídea a la moneda más cercana a la derecha de Mario (`float`)
23. **nearestCoinRight_X:** posición X de la mergeObs de la moneda más cercana por la derecha (`byte`)

24. **nearestCoinRight_Y:** posición Y de la mergeObs de la moneda más cercana por la derecha (`byte`)
25. **numEnemiesObserved:** número total de enemigos identificados en la matriz de observación (`short`)
26. **numCoinsObserved:** número total de monedas identificadas en la matriz de observación (`short`)

Atributos booleanos o binarios

27. **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`)
28. **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`)
29. **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`)
30. **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`)
31. **abyssAhead:** valor numérico binario (0-1) que indica si hay (1) o no (0) foso/abismo/acantilado delante (en matriz de observación [16][10] no hay bloque) (`byte`)
32. **isMarioOnGround:** valor numérico binario (0-1) que indica si Mario está (1) o no está (0) en el suelo (`byte`)
33. **isMarioAbleToJump:** valor numérico binario (0-1) que indica si Mario puede (1) o no puede (0) saltar (bloqueo de salto) (`byte`)
34. **isMarioAbleToShoot:** valor numérico binario (0-1) que indica si Mario puede (1) o no puede (0) disparar (`byte`)
35. **isMarioCarrying:** valor numérico binario (0-1) que indica si Mario está (1) o no está (0) llevando un caparazón (`byte`)
36. **enemyWasKilledBin:** valor numérico binario (0-1) que indica si ha muerto (1) o no (0) un enemigo en la instancia actual (`byte`)
37. **marioWasInjuredBin:** valor numérico binario (0-1) que indica si Mario ha sido herido (1) o no (0) en la instancia actual (`byte`)
38. **isSlopeDown:** valor numérico binario (0-1) que indica si hay (1) o no (0) un desnivel delante (en columna [10]) (`byte`)

Otros atributos

39. **coinsGainedLastTick:** número de monedas conseguidas respecto al tick anterior (`byte`)
40. **marioMode:** modo actual de Mario: 0,1,2 == pequeño, grande, fuego (`byte`)
41. **marioStatus:** estado actual de Mario: 0,1,2 == muerto/derrota, victoria, corriendo/jugando (`byte`)

Acción realizada

ActionKey es el atributo que contiene acción o acciones realizadas durante esta instancia, codificada como un valor numérico que diferencia varias situaciones (`byte`)

0. Ninguna acción o tecla pulsada: **0**
1. Salto (tecla A): **1**
2. Derecha (tecla flecha derecha): **2**
3. Derecha-salto (teclas flecha derecha + A): **3**
4. Derecha-disparar (teclas flecha derecha + S): **4**
5. Derecha-salto-disparar (teclas flecha derecha + A + S): **5**
6. Izquierda (tecla flecha izquierda): **6**
7. Izquierda-salto (teclas flecha izquierda + A): **7**
8. Izquierda-disparar (teclas flecha izquierda + S): **8**
9. Izquierda-salto-disparar (teclas flecha izquierda + A + S): **9**

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 (aunque no tanto como el agente de la práctica anterior, *P2: Aprendizaje basado en instancias*).

Refuerzo

Inicialmente se incluyó el refuerzo en la toma de ejemplos, como atributo final de las tuplas. Este esfuerzo ha sido finalmente ignorado puesto que hemos comprobado en la experimentación y evaluación del agente que la forma en la que lo calculamos originalmente no fue la más acertada. Esta primera aproximación se calculaba de la forma:

$$(intermediateReward\_tickN12 + positionX\_tickN12) - (intermediateReward\_tickN + positionX\_tickN)$$

Para más información sobre como calculamos finalmente el refuerzo, revise la sección *Cálculo del rerfuerzo*.

Toma de ejemplos

Para la toma de ejemplos hemos utilizado tanto el agente humano `P3HumanAgent`, como el agente bot proporcionado `BaseLine`.

El comportamiento humano, en este contexto, realiza acciones que tienen un objetivo constante a lo largo de la partida, no obstante, el humano no es una máquina perfecta y en situaciones idénticas es capaz de tomar acciones distintas, ya que en este contexto su comportamiento es más estocástico que otros bots implementados.

Por esta misma razón, las instancias contendrán variedad de acciones ejecutadas para situaciones similares para las que se ha obntenido distinto refuerzo, y será la tarea del algoritmo de aprendizaje por refuerzo Q-Learning identificar los Q-Valores que determinan que acción es la mejor en cada situación.

En cuanto al comportamiento del bot `BaseLine`, completamente estocástico, nos proporciona multitud de instancias de entrenamiento muy distintas unas a las otras para situaciones muy similares, lo que por los mismos motivos indicados anteriormente, serán fácilmente aprovechables en el algoritmo Q-Learning. Asimismo, el uso del bot nos permite tomar ejemplos con más rapidez que el agente humano.

Por ello, centraremos nuestros esfuerzos en la parametrización y el cálculo del refuerzo incluido en las tuplas (estado actual,acción,estado siguiente) del algoritmo Q-Learning para crear nuestro Agente.

Identificación de situaciones

Como bien se ha comentado anteriormente, las tuplas de datos quedan almacenadas en un fichero `.arff` manipulable con Weka, para que se pueda utilizar esta herramienta para ffiltrar los datos y determinar las diferentes situaciones sin introducir conocimiento del dominio del juego.

Inicialmente se ha utilizado el algoritmo `K-medias` para determinar las situaciones o estados de nuestro agente, ya que es un algoritmo de clustering sencillo y no supervisado, estudiado en las sesiones de teoría, y que nos proporciona unos centroides contra los que comparar posteriormente y con facilidad tanto las instancias de entrenamiento como las instancias del juego durante la ejecución del agente, con el fin de identificar la fila de la tabla Q sobre la que actuar.

A continuación se explica el tratamiento de los datos previo a su clusterización con Weka. Posteriormente se explicarán otros métodos utilizados para la identificación de situaciones y la implementación final.

Borrado de atributos

Las tuplas de datos que disponemos contienen atributos contraproducentes para generar cluster sobre ellos, por lo que en primer lugar, se ha realizado un borrado de atributos antes de calcular los clusters. De esta forma se han eliminado todas los atributos del tick $N+12$, la acción realizada así como los siguiente atributos irrelevantes del tick N:

- timeSpent
- timeLeft
- enemyWasKilledBin
- marioWasInjuredBin
- coinsGainedLastTick
- marioMode
- marioStatus

Estos atributos se han considerado despreciables puesto que no ofrecen información relevante para determinar el comportamiento de mario en el tick actual, por lo que solo serán un inconveniente para determinar los clusters.

Identificación con clustering (K-medias)

Este algoritmo toma `K` como parámetro principal, el cual especifica el número de clusters a formar. Con ello, obtenemos `K` centroides que son representaciones medias de una instancia ideal. De esta forma se ha realizado una experimentación de diferentes número de clusters, para $K = 5$, $K = 10$ y $K = 20$, obteniendo resultados en el comportamiento del agente muy similares para los distintos valores de K con la misma parametrización del algoritmo Q-Learning.

Esto nos ha hecho pensar que el uso de técnicas de clustering no era lo más adecuado en este dominio. Finalmente, hemos procedido a una interpretación de las situaciones a mano, siguiendo las indicaciones dadas por Jose Carlos en clase, para lo que hemos implementado un clasificador manual en forma de árbol `if-else`.

Identificación manual (árbol `if-else`)

Se ha implementado un clasificador manual (sin ayuda de ninguna técnica de aprendizaje automático) en forma de árbol `if-else` para clasificar las instancias en 16 situaciones diferentes. Nos hemos centrado en clasificar las instancias basándonos en si Mario está o no en el aire, si hay enemigos cerca, y si hay obstáculos cerca; en este orden de prioridad. Esto ha dado lugar a las siguientes situaciones:

0. Mario está **en el aire**, hay **enemigo** y **obstáculo** cerca, y **ambos** están **a la misma altura** que Mario.
1. Mario está **en el aire**, hay **enemigo** y **obstáculo** cerca, y el **enemigo** está **a la misma altura** que Mario.
2. Mario está **en el aire**, hay **enemigo** y **obstáculo** cerca, y el **obstáculo** está **a la misma altura** que Mario.
3. Mario está **en el aire**, hay **enemigo** y **obstáculo** cerca, y **ambos** están **a distinta altura** que Mario.
4. Mario está **en el aire**, hay **enemigo** cerca, y este está **a la misma altura** que Mario (no hay obstáculo).
5. Mario está **en el aire**, hay **enemigo** cerca, y este está **a distinta altura** que Mario (no hay obstáculo).
6. Mario está **en el aire**, hay **obstáculo** cerca, y este está **a la misma altura** que Mario (no hay enemigo).
7. Mario está **en el aire**, hay **obstáculo** cerca, y este está **a distinta altura** que Mario (no hay enemigo).
8. Mario está **en el aire** y **no hay enemigos ni obstáculos** cerca.
9. Mario está **en el suelo** y hay un **enemigo** delante **a la misma altura** que Mario.
10. Mario está **en el suelo**, hay un **enemigo** cerca y un **obstáculo a la misma altura** que Mario (posible enemigo encima de obstáculo).
11. Mario está **en el suelo** y hay **enemigos** y **obstáculos** cerca pero **a distinta altura** que Mario.
12. Mario está **en el suelo** y hay **enemigos** cerca **a distinta altura** (no hay obstáculos).

13. Mario está **en el suelo** y hay un **obstáculo a la misma altura** que Mario (que debería saltar, no hay enemigos).
14. Mario está **en el suelo** y hay un **obstáculo** cerca pero **a distinta altura** que Mario (no hay enemigos).
15. Mario está **en el suelo** y **no hay enemigos ni obstáculos** cerca.

Este árbol `if-else` ha sido implementado en la función `calculateSituation()` tanto en el fichero `.java` del agente, como en el fichero `main.java` del código para Q-Learning.

Tras una experimentación, se ha comprobado que los agentes generados para esta identificación de las situaciones, independientemente si habían aprendido de la toma de ejemplos humana o la proporcionada por `BaseLine`, eran ligeramente superiores a los agentes que utilizaban cluster y K-medias, por lo que nos hemos quedado finalmente con esta identificación de las situaciones en forma de árbol `if-else`.

Q-Learning

Aprendizaje de la política de comportamiento

Modificaciones al código proporcionado

El código del algoritmo Q-Learning proporcionado ha sido modificado ligeramente para que sea de aprovechamiento en esta práctica. Además de inicializarse correctamente los valores de alfa, gamma, estados y acciones al comienzo del algoritmo; se han generado las tuplas a partir de las instancias de entrenamientos léidas de un fichero en formato `csv` y se ha calculado el refuerzo de las acciones o transiciones al vuelo.

Para la identificación de las situaciones a las que pertenecían las distintas instancias del juego recogidas en los datos de entrenamiento (tanto para el tick N como para el tick $N + 12$) se ha hecho uso de la función `calculateSituation()` comentada en el apartado anterior *Identificación manual (árbol `if-else`)*.

Asimismo, se han realizado pequeñas modificaciones en el fichero `main.java` y en `QLearning.java` para por una parte evitar que el algoritmo se quedara en bucle infinito intentando encontrar un camino a un estado que no existe, y por otra parte para que imprimiese la tabla Q en un formato directamente aprovechable por nuestro código de Mario, sin necesidad de modificaciones adicionales.

Parametrización del algoritmo Q-Learning

Se han probado diversos valores para los atributos alfa y gamma del algoritmo Q-Learning. Se comenzó inicialmente con unos valores de $\alpha = 0.6$ y $\gamma = 0.8$, que se fueron reduciendo progresivamente.

Pronto nos dimos cuenta que la gran estocaicidad de las acciones, sumado a que requerimos un bot que se centre más en el refuerzo inmediato que no en el refuerzo a largo plazo rquerían unos valores de estos parámetros mucho más bajos de lo que pensamos inicialmente, debido a la funcionalidad de estos parámetros (que ya describimos en el tutorial 4).

Por ello, finalmente hemos utilizado unos valores de $\alpha = 0.15$ y $\gamma = 0.2$ para la implementación de nuestro agente.

En cuanto al número de iteraciones del algoritmo, hemos trabajado durante la evaluación con iteraciones entre 100.000 y 500.000, siempre buscando que fueran superiores al número de tuplas de la toma de ejemplos (~13.000 para la toma humana, y ~250.000 para la toma de `BaseLine`).

El agente finalmente implementado utiliza la toma de ejemplos del bot `BaseLine` y 1.000.000 de iteraciones para el cálculo de la tabla Q.

Cálculo del rerfuerzo

Inicialmente incluíamos el refuerzo al final de cada tupla de la toma de ejemplos, el cual calculábamos durante la toma de ejemplos, pero pronto desechamos esta opción e ignoramos este valor previamente calculado, tras comprobar que la forma de obtenerlo no era la más adecuada.

Tras un periodo de prueba y error, hemos comprobado que la mejor forma de calcular este refuerzo, o la que mejor comportamiento provoca en el agente, es mediante la siguiente fórmula:

$$
\begin{gathered}
reinforcement = \\
0.4 \cdot ((in.intermediaRewardWonFuture6Ticks + in12.intermediateRewardWonLast6Ticks)/2) \\
+0.1 \cdot (in12.intermediateReward - in.intermediateReward)
\end{gathered}
$$

Donde *in* representa el objeto en java de la instancia del tick N de una tupla, e *in12* la instancia del tick $N + 12$.

Una vez calculado el refuerzo de esta manera, si el refuerzo resultante era negativo (algo que personalmente queríamos evitar), procedemos a dividir este entre el valor -100 para convertirlo en un refuerzo positivo pero muy pequeño.

Asimismo, en el cálculo del refuerzo se ha incluido un peso conforme a la acción realizada, para dar más importancia a las acciones que permiten al bot avanzar y completar la máxima longitud posible del mapa, el cual era uno de los objetivos de la práctica. Para ello, elevábamos el valor del refuerzo obtenido hasta un 115 % en el caso de la acción “derecha”, y hasta un 190 % en el caso de la acción “derecha + salto”, valores que experimentalmente resultaron ser los más adecuados.

Tabla Q

La tabla Q obtenida para las instancias de entrenamiento del bot `BaseLine`, para $\alpha = 0.15$, $\gamma = 0.2$ y 1.000.000 de iteraciones es la siguiente. Se ha resaltado en verde oscuro la acción con mayor valor Q para cada estado o situación, y en verde claro la acción con segundo mayor valor Q.

	0 (N)	1 (J)	2 (R)	3 (RJ)	4 (RS)	5 (RJS)	6 (L)	7 (LJ)	8 (LS)	9 (LJS)
S0	69.6	158.57	147.06	244.26	66.81	77.64	137.16	134.47	100.14	78.84
S1	187.13	49.25	196.76	387.95	141.64	156.87	174.08	276.39	125.92	178.81
S2	57.71	106.26	118.54	157.6	54.21	66.2	209.81	115.05	162.35	101.58
S3	123.4	0.0	69.05	0.0	75.93	126.48	0.0	70.24	192.63	76.39
S4	78.1	152.86	117.89	166.29	90.17	71.76	128.17	141.95	91.62	97.98
S5	97.0	61.18	0.0	0.0	132.3	55.07	0.0	0.0	76.5	136.61
S6	145.03	89.91	89.42	176.43	133.59	143.48	86.09	113.65	128.23	130.41
S7	63.78	97.55	93.5	118.83	53.4	57.94	92.25	139.38	58.95	59.42
S8	155.36	73.7	75.56	213.44	171.13	141.87	76.18	46.97	89.74	123.61
S9	79.63	180.35	159.89	186.81	68.41	71.9	126.98	126.88	82.02	95.08
S10	57.34	141.83	111.45	166.61	56.67	80.17	118.02	126.04	72.66	87.15
S11	25.98	0.0	0.0	0.0	25.97	0.0	0.0	0.0	0.0	0.0
S12	110.04	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	128.53
S13	140.49	96.42	105.57	176.54	141.53	135.69	43.48	93.07	125.67	114.49
S14	65.85	150.08	167.29	208.38	78.91	101.17	68.1	72.06	109.19	80.94
S15	64.28	156.37	99.82	158.87	55.79	76.84	138.16	144.47	92.8	90.03

Tabla 1: Tabla Q obtenida para $\alpha = 0.15$, $\gamma = 0.2$ y 1.000.000 de iteraciones (ejemplos de `BaseLine`)

El número que identifica a los estados se corresponde con el indicado en *Identificación manual (árbol `if-else`)*, así como el número que identifica a las acciones se corresponde con el indicado en *Acción realizada*.

Selección de la acción final aprendida

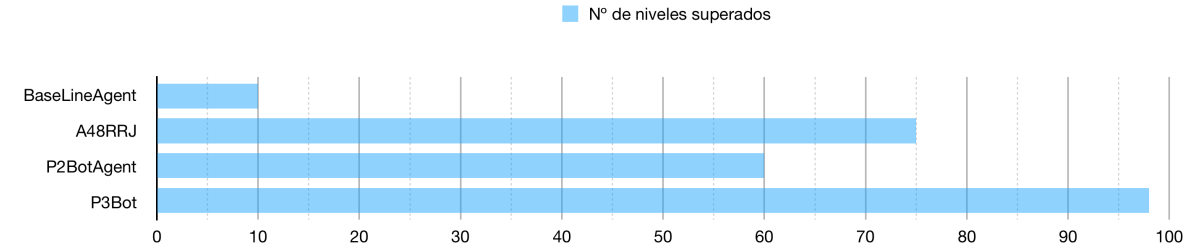
En la ejecución del algoritmo de Q-Learning, se redirigía la salida por pantalla a un fichero con la ayuda de la sintaxis de la terminal bash (entorno GNU/Linux y MacOS). Este fichero se corresponde con `qtable.txt` y era posteriormente cargado en memoria del agente en un array bidimensional, operación que se realiza en el constructor del mismo.

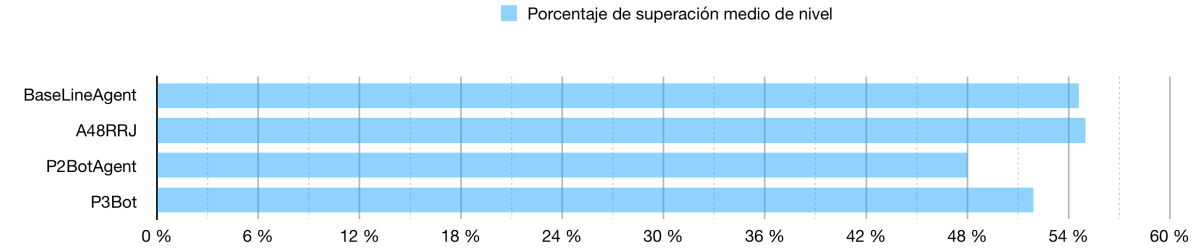
Para cada instancia o tick del juego, se genera su objeto java Instancia y se comprueba con ayuda de la función `calculateSituation()` comentada en el apartado anterior *Identificación manual (árbol `if-else`)* a qué situación pertenece. Esta función nos devuelve el número que identifica la situación a la que pertenece la instancia, por lo que lo único que hay que hacer para decidir la acción a ejecutar es iterar sobre la fila de la tabla Q cuyo índice se corresponda con ese valor y buscar el índice columna (id de la acción) con mayor valor Q.

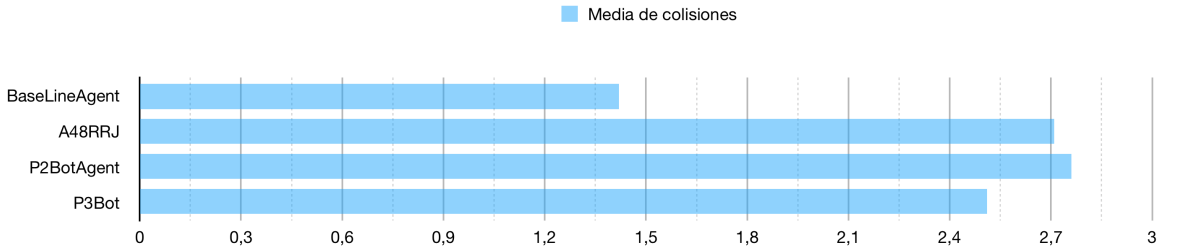
Para evitar quedarnos atascados en **mínimos locales**, el agente implementa un mecanismo con contador en el que si detecta que Mario se encuentra en la misma posición X por más de 12 ticks ejecuta la acción con segundo mayor valor Q, en lugar de la acción con mayor valor Q.

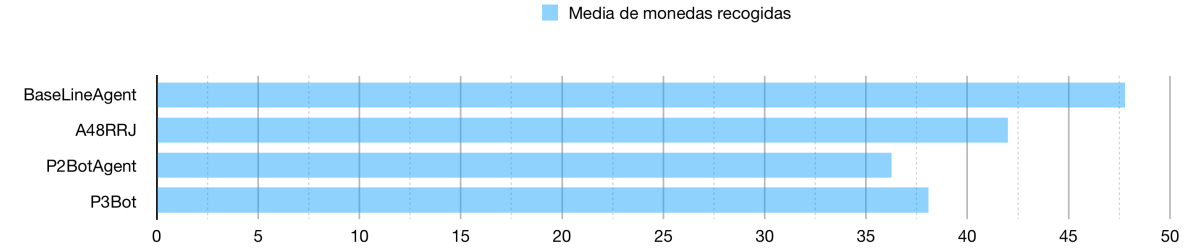
Evaluación

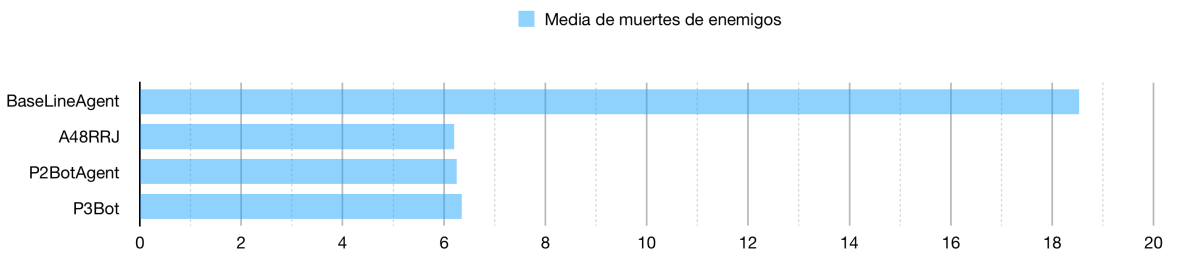
Para la obtención de estadísticas a gran escala (ejecución de 1.000 niveles), hemos utilizado el mismo script en bash que utilizamos durante la práctica 2, `estadísticas.sh`. Este script se incluye junto con el resto de material entregado en esta práctica.


Nº de niveles superados
BaseLineAgent
A48RRJ
P2BotAgent
P3Bot
0
10
20
30
40
50
60
70
80
90
100


Porcentaje de superación medio de nivel
BaseLineAgent
A48RRJ
P2BotAgent
P3Bot
0 %
6 %
12 %
18 %
24 %
30 %
36 %
42 %
48 %
54 %
60 %


Media de colisiones
BaseLineAgent
A48RRJ
P2BotAgent
P3Bot
0
0,3
0,6
0,9
1,2
1,5
1,8
2,1
2,4
2,7
3


Media de monedas recogidas
BaseLineAgent
A48RRJ
P2BotAgent
P3Bot
0
5
10
15
20
25
30
35
40
45
50


Media de muertes de enemigos
BaseLineAgent
A48RRJ
P2BotAgent
P3Bot
0
2
4
6
8
10
12
14
16
18
20

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>
```

- ...

Conclusiones

De la realización de un agente inteligente que utiliza *Q-Learning* podemos concluir lo siguiente:

- a
- b
- c

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.